Concurrent Embedded Design for Multimedia: JPEG encoding on Xilinx FPGA Case Study

Jike Chong
Abhijit Davare
Kelvin Lwin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-40
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-40.html

April 16, 2006

Acknowledgement

The authors would like to thank Kaushik Ravindran, Yujia Jin, N.R. Satish, and Douglas Densmore for insightful discussions on the capabilities of the Xilinx Platform.

Concurrent Embedded Design for Multimedia: JPEG encoding on Xilinx FPGA Case Study

Jike Chong, Abhijit Davare, Kelvin Lwin
{jike, davare, klwin}@eecs.berkeley.edu

***Abstract*— Parallel platforms are becoming predominant in the embedded systems space due to a variety of factors. These platforms can deliver high peak performance if they can be programmed effectively. However, current sequential software design techniques as well as the Single Program Multiple Data (SPMD) programming models often used in the High Performance Computing (HPC) domain are insufficient. In this report, we experiment with a dataflow programming model for multimedia embedded systems. By applying this programming model to a common application and embedded platform, we get a better idea of the implementation challenges for this class of systems.**

I. Introduction

In the past, the programming of concurrent systems was a challenge primarily limited to the HPC domain. Today, highly concurrent programmable platforms are becoming widespread in the embedded systems domain as well, and due to the unique verification and synthesis requirements for this domain, the programming challenges are significant.

With the continuing availability of additional transistors at every process generation, designers have a choice when migrating their designs. The first option is to leave the design functionality unchanged and take advantage of the benefits of smaller dies in the subsequent process generation. This is by far the easiest option. Unfortunately, it is not practical in many instances since the system requirements routinely increase along with the availability of additional transistors.

The second alternative is to redo the hardware design to support the additional functionality by building a more complex gate level netlist. For instance, a 32-bit RISC core can be enhanced with better branch prediction or a re-pipelined to achieve a higher frequency to offer higher performance. This approach was taken in the past and allowed software to remain unchanged across process generations. However, there are two main problems that have emerged with this approach. First, the design effort involved in verifying the new design is substantially more than in the past due to increase in design complexity, manufacturing variation and other nanometer fabrication issues. The second problem is that power consumption vastly increases when trying to run more transistors at greater frequencies.

The third alternative is to abandon the uniprocessor paradigm and place multiple general purpose or specialized cores on chip. This has the advantages of easing the hardware design and integration issues. However, the sequential software paradigm is no longer valid. The onus is placed on the software developer to effectively utilize the additional parallelism provided by the hardware. Unfortunately, tools for effectively transforming sequential software into concurrent software do not exist.

The hypothesis of this research is that appropriate concurrent programming models must be developed to effectively utilize future multi-core embedded platforms. In this paper, we concentrate on experimenting with a dataflow model for multimedia applications. We show that the choice of this application model allows us to quickly create feasible implementations while still retaining a clear path to higher performance implementations.

II. Embedded Systems vs. HPC

The HPC domain differs in several ways from the embedded systems domain. In this section, these differences will be highlighted according to several different categories.

A. Platforms

As compared to HPC systems, embedded systems platforms have a number of unique characteristics.

First, the balance between computation and communication is currently much better in embedded systems than in HPC. Since the multiple cores in embedded systems are on the same chip, access to memory resources is relatively faster. Embedded applications typically have stringent cost or power requirements that prevents the use of large memories. The same requirements also prevents the use of very high frequency cores. The practical consequence is that the amount of computation needed to justify a communication operation is much lower in embedded systems. Of course, the trend is that communication will become relatively more expensive in the future.

In HPC, the programming model is usually SPMD, where a single program is distributed to multiple processing elements. In embedded systems, a MPMD paradigm exists, where certain components of the overall computation are assigned to specialized processing elements. The verification challenges increase when the original code is fragmented and distributed.

Unlike general purpose processors in HPC, each embedded processor can have different functional units that can better execute certain instructions. As an example, without a barrel shifter in an uBlaze, multiply can be performed faster than a shift. The designer not only has to carefully tune the code in order to achieve performance but must choose or optimize the type of functional units available in each of the uBlazes as shown in Figure 1. An easy solution is to instantiate all

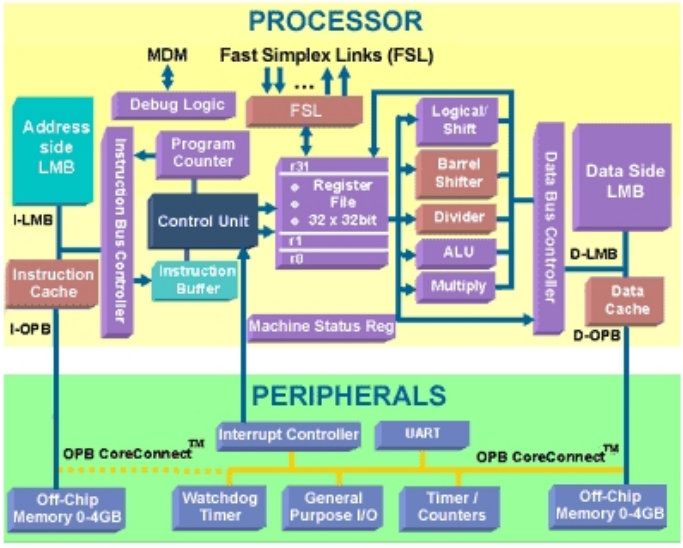

Fig. 1. MicroBlaze Soft Processor Block Diagram [1]

available functional units for each of the ublazes. However, this has a direct impact on the number of ublazes that could be instantiated on the same FPGA.

The desire in embedded systems is to obtain predictable performance from the application. Average case performance is not as important as worst-case performance. Therefore, embedded platforms typically do not have complex memory hierarchies seen in HPC nodes. Another reason to avoid complex pipelines and memory hierarchies is power consumption. Typically, the worst-case joules per instruction value cannot be improved by making each processor more complex.

Finally, memory coherency is another area where differences exist between HPC and embedded design. Significant overhead is required to maintain coherency over distributed memory. In HPC, coherency is used to expand the reach of legacy sequential programs. Embedded systems typically can not afford such overhead. Embedded applications in the multimedia domain typically require only a small data workset at any given time, distributed fast memories are capable of implementing the applications, but are harder to program.

B. Applications

Application development in embedded systems has typically been carried out either in assembly language or low-level C. Object orientation and memory allocation/deallocation have typically not been used due to the overhead. Unfortunately, this means that software reuse is typically difficult, especially for larger designs. With concurrent designs, this problem is exacerbated.

Since verification is important component of the embedded design process, the code that is deployed on these systems needs to be analyzed for correctness. Even if code snippets are verified manually, their composition on a multiprocessor implementation is still difficult to verify.

Experience in HPC shows that performance is directly related to the quality of the code generated with many optimizations not performed by even high-power compilers today. Even for uniprocessor code, the inner loop optimization is one of the key to achieving high performance by keeping the functional units operating at peak utilization. The user has to be highly knowledgeable and give as many hints to the compiler as possible to get decent performance. The situation is only worse in multiprocessor systems where communication has to be taken into account. In the embedded domain, compilers are lightweight and typically implement only a fraction of commonly applied optimizations. In addition, the paucity of instruction memory on embedded processors implies that some optimizations may not be feasible. For instance, a uBlaze soft processor can support from 4 KB to 64 KB of RAM for both instruction and data. So the designer must be careful in using scarce resources for optimal performance.

C. Systems

The primary focus of HPC systems is simulation or offline data analysis. Neither of these needs to be carried out in real time. Instead, the objective is to complete the simulation or analysis run as quickly as possible. Therefore, the design of these systems concentrates on average-case performance. By exploiting spatial and temporal locality in the applications, the system will complete runs quicker on average. In the worst case, the simulation or analysis may take substantially longer, but these occurrences are relatively rare and not considered in the design phase.

By definition, the systems we are considering are embedded in the environment around them, they must function according to the latency, throughput, energy and power characteristics that the environment features. This implies that in order to ensure that an embedded system will function correctly in its environment, certain worst-case requirements need to be met. The design process must ensure that for all inputs, the system will meet these worst-case requirements. In order to efficiently carry out this type of analysis, embedded systems design focuses on restricted programming models, or models of computation, that can be verified with respect to the requirements.

Another difference between HPC and the embedded systems domain has to do with the market for these systems. The users of HPC systems typically want to carry out cutting-edge science. Typically, the cost, resources, and infrastructure required to maintain a high-performance computing facility make it a viable option only for governmental or large academic institutions, but relatively few industrial customers. Even when HPC is used in industrial contexts, it is used in the back-end to carry out offline activities. Applications for these HPC platforms are usually written by scientists, not by programmers. Therefore, there is an effort made to provide and support appropriate programming models (MPI, OpenMP, Pthreads, etc) so that applications can better exploit a larger fraction of the peak capacity. Also, the application codes that run on HPC platforms are usually long-lasting, having been developed relatively slowly over a period of years.

In the embedded systems domain, the situation is very different. Typically, embedded systems development is carried out in an industrial setting by a wide variety of companies. Due

to time-to-market pressures, there is typically no particular methodology applied to embedded software development, it is simply the usage and adaptation of existing low-level code. Assembly and C are the predominant development languages, any layers on top of these do not enjoy widespread acceptance. This approach is no longer sustainable with highly parallel platforms.

III. Contributions

The main contributions of this research are the performance-oriented characterization of the Virtex II Pro FPGA platform and the development of a lightweight environment for dataflow application modeling.

A. Performance Analysis of Platform

One of the main contributions of this work is the characterization of a uBlaze soft processor and FIFO-based platform. The characterization relates to the inter-processor communication and the computation cost on each uBlaze. The relationship between logic utilization, FIFO depth, and FIFO latency is the most important relationship for communication. For computation, the inclusion of specialized uBlaze resources such as barrel shifters and multipliers, and their influence on runtime of code as well as logic utilization was understood.

B. Pthreads Dataflow Modeling

The datraflow [2] paradigm is effective for distributed data-streaming applications. Unfortunately, existing software frameworks that can help model dataflow applications such as Ptolemy II [3] and Metropolis [4] are typically heavyweight and not suited for quick migration to embedded processors.

The requirement for a lightweight dataflow modeling environment allows a program with multiple processes to be specified in C-language code that can directly be implemented on the target processor. Pthreads [5] provides the ability to implement multi-process C programs with minimal modification to the final C code. To implement dataflow semantics, we implemented a FIFO class for inter-process communication with bounded storage, blocking reads and blocking writes.

IV. Dataflow

In this section, the programming model chosen for this case study – dataflow – along with the related Process Networks model, will be described in further detail.

Kahn Process Networks [6] is a model of computation where concurrent processes communicate with each other through point-to-point one-way FIFOs. Read actions from these FIFOs block until at least one data item (or token) becomes available. The FIFOs have unbounded size, so write actions are non-blocking. Reads from the FIFOs are destructive, which means that a token can only be read once. The appealing characteristic of the KPN model of computation is that execution is deterministic and independent of process interleaving. Also, this model of computation (MoC) allows natural description of applications since it places relatively few requirements on the designer other than blocking reads.

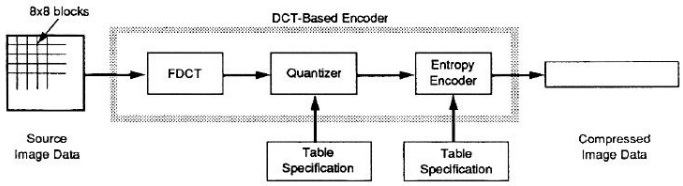

Fig. 2. JPEG encoder block diagram

Dataflow process networks are a special case of Kahn Process Networks where the execution of processes can be divided into a series of atomic firings [2]. This MoC in general suffers from the same undecidability as Kahn Process Networks [7]. In static dataflow [8], the number of tokens produced and consumed for each firing is statically fixed. Due to this restriction, aspects such as scheduling and buffer size can be computed statically and efficiently. The key limitation is that data-dependent behavior is difficult to express. This limitation makes this MoC unsuitable for many practical applications. Related work including Cyclo-static Dataflow [9], Heterochronous Dataflow [10], and Parameterized Dataflow [11] attempt to extend static dataflow but retain decidability by allowing structured data-dependent behavior.

V. Application and Platform

In this Section, the JPEG encoder application and the Xilinx Virtex II FPGA platform will be described.

A. JPEG encoder

The JPEG encoder [12] application, Figure 2, is required in many types of systems, from digital cameras to high-end scanners. The application compresses raw image data in 4:4:4 format as per the JPEG standard and emits a compressed bitstream. This application was chosen since it is relatively simple, yet is a representative component of a wide class of multimedia applications. The main blocks in the JPEG encoder algorithm are utilized in several video compression algorithms including MPEG-2 and the next generation H.264 standard.

In the preprocessing step, the raw RGB image data is first converted into YUV format, which represents the image as a set of luminance, blue chrominance and red chrominance components. This is advantageous for compression since the human eye is more sensitive to luminance than either of the chrominance components. The chrominance components can therefore be compressed further than the luminance component.

Next, each of the three components is converted into 8x8 blocks and processed independently. First, each 8x8 block passes through a forward DCT block, which converts the spatial data in the block into frequency data. This step in the flow does not result in the loss of any information, besides round-off errors. Next, the DCT outputs are quantized, or divided, by coefficients from a user-defined table. The quantization step is the fundamental information-losing step in the compression process and attempts to reduce many of the higher DCT frequency coefficients to zeros.

Then, run-length encoding and Huffman encoding are carried out on the quantized coefficients to reduce the number of bits needed to represent them. The Huffman compression tables are hard-coded and supplied by the user. The JPEG file consists of the user-supplied tables and the compressed image bitstream.

B. Xilinx Platform

The Xilinx ML310 is a development board for a Virtex-II Pro XC2VP30-based embedded system. In addition to more than 30,000 logic cells, over 2,400 Kb of BRAM, and dual PPC405 processors available in the FPGA, ML310 provides onboard Ethernet MAC/PHY, DDR memory, multiple PCI slots, and standard PC I/O ports within an ATX form factor board. An integrated System ACE CF controller is deployed to perform board bring-up and to load applications from the included 512MB CompactFlash card.

The programmable logic cells on the FPGA can be used to implement uBlaze soft processor cores, which can be connected using a variety of communication channels. Choices of communication channels include system peripheral buses and hardware FIFOs such as the Fast Simplex Links (FSL), which are direct communication channels to and from architectured registers in the soft processors.

The uBlaze 32-bit soft processor is a standard RISC-based engine with a 32 register by 32 bit LUT RAM-based Register File, with separate instructions for data and memory access. It supports both on-chip BlockRAM and/or external memory. All peripherals including the memory controller, UART and the interrupt controller run off of the On-chip Peripheral Bus (OPB). Additional processor performance is achieved by utilizing fast hardware divide and hardware multiply capability associated with the dedicated 18 bit x 18 bit multiplier block.

The uBlaze requires 950 logic cells on the Virtex-II Pro, and supports a variety of communication channels such as OPB and Fast Simplex Link(FSL). The FSL has its own interface with the architectural registers, which bypasses the slower memory controllers and the OPB. There can be up to 8 input and 8 output FSLs per uBlaze, each of which can be considered an unidirectional FIFO.

VI. Design Space Exploration

For design space exploration (DSE), we start from a feasible point within the design space, and move to "nearby" (incremental changes) feasible points such that each move results in a better objective value. At the end, we may be trapped in a local minimum, which may or may not be the global minimum. Since we do not have a pre-existing characterization of the design space, we can't apply a global optimization technique, but this initial exercise will allow us to capture the features of the design space for future automation.

DSE involves re-partitioning the design to exploiting actor-level parallelism in the application while maintaining functional correctness. The re-partition should result in an improvement in the objective value. Realizing this approach on an embedded platform requires the ability to debug and characterize a design effectively.

On the FPGA platform, to realize and measure a multi-processor configuration, we instantiate a multi-processor topology with uBlaze, FSL, OPB, UART and timer peripherals, implement and debug required application on the system, and acquire the necessary statistics to point to possible modifications.

In instantiating multi-processor topologies, the size of a multi-processor configuration is not a direct indication of complexity of realization. A 9 uBlaze 3x3 torus configuration takes only 83 minutes to place and route, where as an 8 uBlaze configuration in a four stage pipeline with three uBlaze's in parallel in the 2nd/3rd stage takes 1,291 minutes to place and route. The efficiency of realizing a regular configuration on the FPGA fabric prompted us to use a regular torus structure for functional debugging, and later pruning these structures to specialized topologies for the area related objective evaluation. Note, using the 9 uBlaze 3x3 torus to implement an 8 uBlaze configuration is not without penalties. To accommodate all links in a torus structure, global routes through these platforms may compromise system performance. One such example is illustrated in section VIII.

The Xilinx uBlaze debugging interface we utilized requires a port for each uBlaze as well as an UART device on the On-chip Peripheral Bus (OPB) to pass any output to the serial port. In a multi-uBlaze implementation, such debugging interface imposes significant limitations on what can be observed on the serial port. Since access to the UART is arbitrated on a byte-by-byte basis, multiple output UART requests from uBlaze's to the bus will render a set of outputs unintelligible.

An arbiter can be implemented to grant access to each uBlaze as it requests usage of the output bus. However, such mechanisms induce significant overhead in resources and performance, a simplified version also limits the scalability of the solution. Instead, we leverage the ease of functional debugging in the Pthreads environment and map a set of functionally correct partitions one-to-one to a topology in uBlaze's and FSLs. Then, only minor checking is required on the FPGA, and it can be performed on one uBlaze at a time to avoid conflicts on the OPB. Thus, in implementing the required application, the Pthreads code is mapped directly to the multi-processor configuration after functional verification has taken place.

The performance of the design is measured by checking a Xilinx Timer instance on the OPB. The timer increments at the input clock rate, making it effectively a free-running cycle counter. Multiple requests to the timer on the OPB are not arbitrated, so only one uBlaze can be timed in any given run. In acquiring the necessary statistics, timer access from a ublaze through the OPB requires approximately 40 cycles of overhead per access. Since timing statistics are measured in 100,000 cycle range, this overhead is negligible. Analyzing relative performance of different partitions will point to bottlenecks in the implementation, where parallelized tasks may require balancing, or additional parallelism may be required.

VII. Traversal of Design Space

The objective for this case study is to optimize JEPG performance of one encoding stream within the area budget

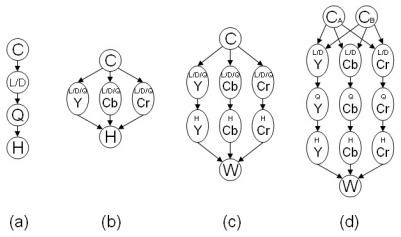

Fig. 3. Sequence of topologies experimented with in design space exploration

of the 2VP30 chip. The baseline design is a single uBlaze implementation, where the entire algorithm is implemented on a single processor with appropriate instruction and data memory. The topologies traversed from this initial implementation are summarized in Figure 3.

There are two natural ways to parallelize the JPEG algorithm, one based on pipelining the processing steps as shown in Figure 3a, the other based on independent processing of each channel as shown in Figure 3(b). Analysis of Figures 3(a) and 3(b) indicates that the time taken for each part of the algorithm differs drastically. The blocking read and write semantic on the FIFOs self-times the steady state throughput to the worst case execution time of all pipeline stages. In the case of implementation (b), the bottleneck stages are level-shift and DCT.

Implementation (b) improves the throughput by recognizing that large portions of the three channels in each MCU can be processed in parallel. By working on all the processing steps of each channel, we evenly divide up the work into three parts, thus reducing by a factor of three the average execution time for the processes. The bottleneck in implementation (b) is at the Huffman stage, where run length encoding and Huffman encoding are done serially for the three channels. The reason for this serialization is the usage of a simplified version of the compression code to transform a fixed length character stream into a variable length bit stream.

In (c), run length encoding and Huffman encoding are separated out into three concurrent stages, one for each channel. This operation eliminates the bottleneck in the pipeline, but adds the overhead of managing the convergence of three variable length bit stream in the final write stage. The bottleneck is then shifted to the level-shift, DCT and quantization stage.

In (d), the bottleneck in (c) is separated into two stages, the level-shift/DCT, and the quantization stage. This alone will shift the bottleneck to the first stage - the color conversion stage. The color conversion is responsible for producing the three parallel channels for the rest of the algorithm to work on, and needs to supply all three pipelines. However, there is exploitable parallelism among the MCUs to be converted. The color conversion process is distributed over two uBlazes, each of which communicate with all three color channels. The bottleneck is then shifted back to the level-shift/DCT stage, with all stages being roughly balanced.

Throughout this exploration procedure, to obtain finer grain parallelism, the code was optimized with respect to the uBlaze it was being executed on. Loop overhead in tight loops causes a lot of wasted cycles. Overhead is reduced by unrolling loops in the critical stages to increase performance. The loops in the non-critical stages should remain unchanged to maintain code density. The uBlaze's in critical stages are also fitted with accelerators such as barrel shifters to boost performance.

When the area occupied by the design comes too close to the capacity of the FPGA, system performance is lost due to congestion.

VIII. Experimental Results

For the JPEG application, the quantization and huffman tables were provided according to the reference implementation of the standard. The test image in raw RGB format was preloaded onto the FPGA, and the result was written out to memory.

The logic slice usage on the FPGA, timing and performance numbers are presented in Table I.

TABLE I
Experimental Results

Topologies	Area[1]	Clk[2]	Clk[3]	Cycle	Performance
Single mB	10%	100	100	595,525	1
(a) 4 mB	26%	100	100	291,470	2.04
(b) 5 mB	32%	100	100	224,506	2.65
(c) 8 mB [4]	64%	73.8	100	112,093	5.31
(d) 12 mB [5]	76%	82.3	100	62,084	9.59

[1] Percentage slices used in a Xilinx 2VP30 FPGA
[2] MHz achieved with global debugging bus
[3] MHz achieved with only point to point communication channels
[4] implemented on top of 9 mB torus
[5] extended from 9 mB torus

The uBlaze is a modular building block that is capable of running at 122Mhz individually. When building a system of many uBlazes, OPBs and FSL are used as communication channels between the uBlazes. We observe that the overall system performance is highly dependent on the communication channel implementation. In the implementations where we link uBlazes (up to 6) with an OPB bus to the RS232 debugging interface and timer(Clk[2] column in Table I), the global bus degrades system performance significantly for larger designs.

One may note that in row (c) and (d) of Table I, performance does not degrade monotonically with respect to logic slice usage for designs with a global debugging bus. This is a serious concern for system level design space exploration as the performance of FPGA implementations is heavily dependent on placement and route and overall architecture.

For debugging the design, the OPB is used to monitor variables on one uBlaze at a time. After the debugging process is complete, the application only requires point to point links to operate and the OPB is no longer required for all but one uBlaze to monitor performance. As seen in (Clk[3]) column of Table I, The system can operate at 100MHz with the OPB restricted to serving only one uBlaze, thus the system performance bottleneck is removed.

The relative efficiency graph in Figure 4 shows the performance achieved vs. FPGA fabric utilization for the different topologies. The key message from this data is that effective use of the available chip area has a substantial impact on total

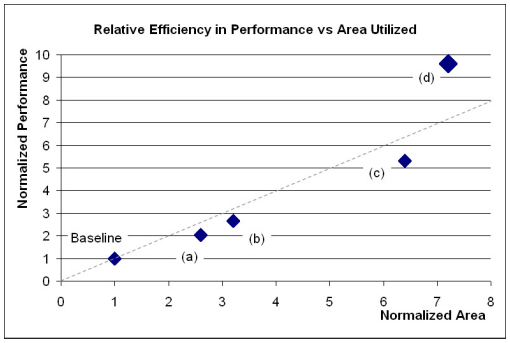

Fig. 4. Relative area-performance efficiency of various topologies

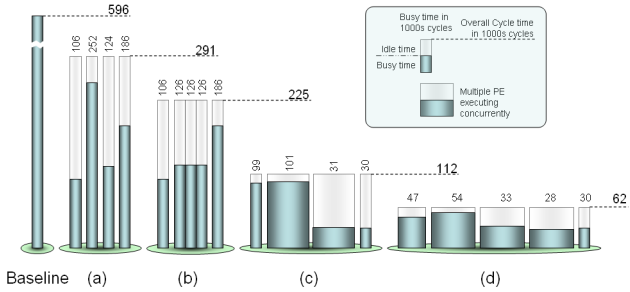

Fig. 5. Runtime analysis of various topologies

performance. Super-linear speed up was achieved when the system overhead is amortized over 12 uBlazes in topology (d).

The efficiency for topologies (a) and (b) is compromised by an imbalance in the pipeline, as illustrated in Figure 5. In this figure, the balancing of the pipeline stages for each of the different topologies is detailed by showing the busy and idle times for each processor. The total number of cycles taken to process the image is also shown.

IX. Conclusions

In this work, we have applied a dataflow programming model to a case study from the multimedia domain. We have characterized several aspects of the application and platform and demonstrated the ability to manually traverse the design space and arrive at a competitive solution.

The most important lessons from this exploration procedure are as follows. First, we discovered dataflow is an efficient architecture for system partitioning in FPGA based designs. The point to point communication channels are efficiently mapped by the backend tool flow to achieve high clock frequency. Global structures such as debugging buses are essential during the development phase of a system, but severely limit system performance. Removing global debugging buses once the system is debugged recovers the achievable performance. Super-linear area-performance efficiency increase can be achieved by amortizing system overhead over up to 12 uBlazes, allowing this parallelization approach to gain almost an order of magnitude of performance. Second, we determined that for this algorithm, large FIFOs are only required to eliminate the effects of jitter on overall performance. If a certain stage in the algorithm has data dependent complexity, then having large input or output FIFOs will prevent the variation from causing upstream and downstream processors to block. However, no FIFO size can overcome the effects of an imbalanced pipeline. Ideally, analysis of the dataflow graph should be able to identify minimum lengths on the FIFOs to avoid deadlock whereas simulation should help identify the extent of data-dependent computation in each stage.

X. Future Work

This research opens up a number of avenues for future research. First, automated synthesis techniques can be applied to automate the design space exploration process. Second, additional capabilities of the FPGA platform can be utilized in an effort to improve overall performance.

If certain restrictions are placed on the communication patterns of the actors in the dataflow diagram, then design properties such as the amount of FIFO memory needed and the schedules on individual processors can be statically determined.

The Virtex II FPGA platform has several other components that were not utilized in this work. For instance, the PowerPC cores offer four times the performance of the uBlaze's without any additional area penalty. Also, bus-based interconnect structures are available on this platform. Additional cores such as the picoBlaze soft processor are also available. These cores use a small fraction of the area required for a uBlaze, but are 8-bit and have substantially lower processing capabilities. Finally, we would like to better characterize the performance of our implementations on the platform by looking at the power consumption.

References

[1] Xilinx inc website. www.xilinx.com.

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

[3] Xiaojun Liu, Yuhong Xiong, and Edward A. Lee. The Ptolemy II Framework for Visual Languages. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 50. IEEE Computer Society, 2001.

[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, April 2003.

[5] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

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

[7] Joseph Tobin Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical Report ERL-93-69, 1993.

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

[9] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cyclostatic dataflow, 1995.

[10] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999. Research report UCB/ERL M97/57.

[11] B. Bhattacharya and S. S. Bhattacharyya. Parameterized modeling and scheduling of dataflow graphs. Technical Report UMIACS-TR-99-73, Institute for Advanced Computer Studies, University of Maryland at College Park, December 1999. Also Computer Science Technical Report CS-TR-4083.

[12] Gregory K. Wallace. The JPEG still picture compression standard. 34(4):30–44, April 1991.