# GENERATING INFRASTRUCTURE FOR FPGA-ACCELERATED APPLICATIONS

*Myron King, Asif Khan, Abhinav Agarwal, Oriol Arcas, Arvind*
{mdk,aik,abhiag,arvind}@csail.mit.edu, oriol.arcas@bsc.es
Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory
Universitat Politecnica de Catalunya - BarcelonaTech, Barcelona Supercomputing Center

## ABSTRACT

Whether for use as the final target or simply a rapid prototyping platform, programming systems containing FPGAs is challenging. Some of the difficulty is due to the difference between the models used to program hardware and software, but great effort is also required to coordinate the simultaneous execution of the application running on the microprocessor with the accelerated kernel(s) running on the FPGA.

In this paper we present a new methodology and programming model for introducing hardware-acceleration to an application running in software. The application is represented as a data-flow graph and the computation at each node in the graph is specified for execution either in software or on the FPGA using the programmer's language of choice. We have implemented an interface compiler which takes as its input the FIFO edges of the graph and generates code to connect all the different parts of the program, including those which communicate across the hardware/software boundary. Our methodology and compiler enable programmers to effectively exploit FPGA acceleration without ever leaving the application space.

## 1. INTRODUCTION

The use of specialized hardware by an application running in software can sometimes improve its performance or lower its power consumption. Using traditional methodologies, the task of migrating functionality from software (SW) to hardware (HW) can be characterized as follows:

- Profile the application and select the kernel to be re-implemented in HW.
- Define the interface, and implement the specialized HW.
- Translate data-types which cross the HW/SW boundary.
- Implement additional HW and SW to connect the application components to the communication fabric.
- Restructure the application SW to more effectively exploit the HW accelerator.

While some of these steps require deep insight into the organization of the algorithm, others are simply tedious and error prone. Our experience has led us to a new methodology that addresses some of the problems encountered in traditional methods. We begin by using data-flow to describe the high-level organization of each application. The computation at each node in the data-flow graph is implemented in the programmer's language of choice, and a target for the node (either HW or SW) is also specified.

By enforcing data-flow as the only method of communication between HW and SW, we separate the task of implementing the communication into two parts. At the application level, the programmer simply defines the HW/SW interface as the union of all data-flow edges which cross the HW/SW boundary. At the systems-level, we consider the separate problem of efficiently mapping the FIFO channels to the physical platform.

With a target designated for each node in the graph, an interface compiler we have implemented specifically for this task compiles the FIFO edges and generates code to connect the nodes. Adjacent nodes assigned to SW communicate using shared memory, while adjacent nodes assigned to HW are connected using efficient FIFOs implemented in the FPGA fabric. Edges crossing the HW/SW boundary require substantially more work, but on supported platforms these too can be *automatically* implemented on the shared communication fabric, along with the SW drivers and HW shims connecting the accelerator to the rest of the system. Assuming the efficient and automatic implementation of the communication model, the primary challenge for the application programmer is now to enable the appropriate granularity of communication.

**In this paper, we present a new methodology and programming model for HW/SW communication based on data-flow, and show how it is implemented by our interface compiler. Through the use of examples, we demonstrate the simplicity and flexibility of our approach and the efficiency of the generated implementations on the Zynq platform from Xilinx.**

**Paper Organization:** Section 2 illustrates the steps enumerated in the Introduction, as well as other challenges intrinsic to the use of specialized HW. In Section 3, we introduce the programming model for HW/SW communication, discuss related works and the potential for automation, and present the compilation strategy by which the interface compiler implements the programming model. We provide an evaluation in Section 4, and conclude with Section 5.

## 2. A MOTIVATING EXAMPLE

In this section we describe a structured approach to the task of accelerating a C++ implementation of Ogg Vorbis, an open-source audio CODEC designed for low-complexity decoding. We have chosen this example for its relative simplicity, but the complications which arise are only exaggerated as the applications grow in complexity.

**Kernel Selection:** Figure 1.a shows the organization of a single-threaded C++ implementation of the Vorbis CODEC. Using profiling tools, we observe that most of the execution time is spent computing an Inverse Fast Fourier Transform (IFFT) ❶. This "hot spot" is a natural candidate for FPGA

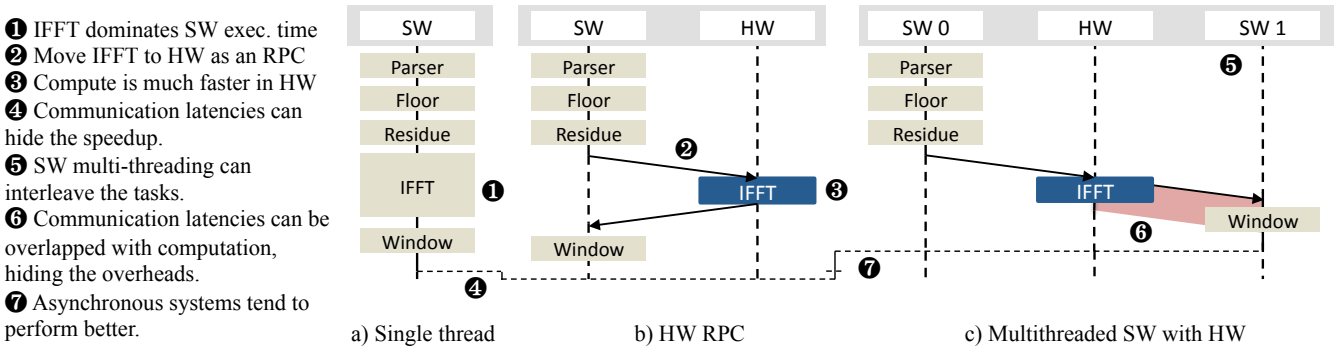



**Fig. 1**. The organization of the application evolves as we introduce hardware and then optimize its performance.

acceleration.

**Defining the Interface:** The IFFT interface ❷ we choose must reflect the flexible nature of the application, which can accommodate a range of frame sizes. Flow control is also necessary to enforce correct use of the accelerator. Our HW implementation of IFFT will have three logical ports: Audio frame size, data input, and data output. For each logical port, we will assign thirty-two bits for the data and two additional *ready* and *enable* bits for flow control. On each port, the same hand-shaking protocol is implemented to apply back-pressure and maintain flow-control. For example, if the "data input ready" signal is high, the next input word can be transmitted by asserting "data input enable". The interface and associated timing properties are quite conventional from a hardware perspective, but making the connection from this interface to a function call in SW will require substantial effort.

**Data-Type Conversion:** Mismatch of data representations is a common source of errors. Objects generated by the application must be correctly converted to the bit representation expected by the accelerator. Verilog and C++ compilers may have different endian conventions, and there are SW structures such as pointers which cannot be directly represented in HW. Maintaining consistency between two separate representations of the same datatypes is both tedious and error-prone.

**Implementing the specialized HW:** In general, specialized HW tends to be more efficient, in terms of energy or time, than SW implementations. This effect is especially acute for highly parallel algorithms, like IFFT, which will perform faster on an FPGA than in SW ❸.

**Connecting To The Communication Fabric:** Tight FPGA/Microprocessor integration is generally achieved using some kind of bus. The hand-shaking protocol which we implemented for the IFFT accelerator is similar to many bus protocols, so we can simply expose the device to the bus using three separate slave interfaces. We will invariably need to implement a thin layer or shim to sit between the IFFT HW and the bus to correctly convert between the two protocols.

The next task is to expose the HW functionality to the SW. What happens if SW attempts to read or write a port whose Rdy signal is not high: should these invocations block or should we expose this in some other way? Most bus protocols support blocking read/write functionality, which we can use to expose each of the logical HW ports as blocking remote procedure calls (RPC) to the application SW. With a suitable SW abstraction of the FPGA accelerator, we can integrate it into the application code as shown in Figure 1.b. Given the latency introduced by transferring the data to and from the accelerator over the bus one word at a time, it is highly unlikely that the overall throughput of the implementation will improve by using the accelerator in this manner ❹.

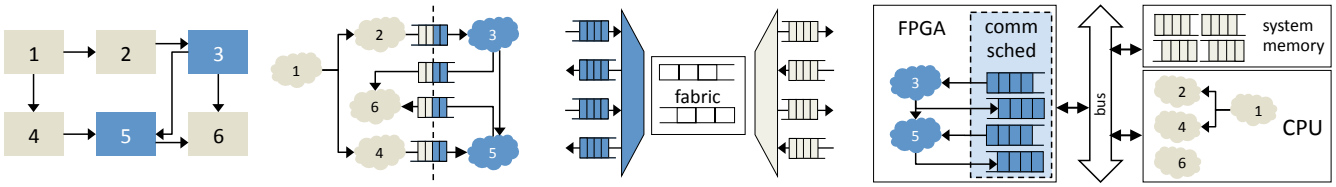
Performance bottlenecks are not the only motivation for specialized hardware: in the cases where it does not dramatically improve throughput, the result may be computed using far less energy. Additionally, the cost of data-transfer in terms of energy must also be taken into consideration when selecting the accelerated kernel.

**Restructuring The Application SW:** Where significant data transfer is involved, it is important to find ways to speed up the communication. It is often impossible to change the *latency* of communication, which is tied to system components like bus and network. Several potential solutions exist to hide this latency, chief among which are increasing the granularity of communication, and introducing pipelining.

The cost of a bus transaction can be amortized by using DMA hardware to transfer larger blocks of data directly into the accelerator memory, but care must be taken to ensure that sufficient buffering exists before the burst transfer is initiated. In the application SW, substantial reorganization may be required to increase the communication granularity ❺. On some systems, failure to ensure these buffering conditions raises the possibility of deadlock: if the HW accelerator exerts back pressure during a burst transaction, the DMA engine will stall until the HW begins accepting tokens again. If bus control cannot be transferred, the entire system may deadlock if the HW cannot make forward progress.

Instead of modifying the IFFT implementations to guarantee sufficient buffering, we choose a simpler approach in which we add a thin layer, or shim, on top of the accelerator interface consisting of buffering and some control logic to ensures that transactions will begin only when certain conditions are met.

Pipelining the HW accelerator hardware can further improve the throughput ❻. This requires substantial transformation of the HW implementation, but once completed we can exploit this concurrency in SW through the use of multi-threading, as shown in Figure 1.c. Instead of block-




a) Data-flow/Kernel Selection b) Interface Definition c) Infrastructure Synthesis d) Mapping to Platform

**Fig. 2**. With HW/SW communication expressed as data-flow, our interface compiler can automate b), c), and d).

ing function calls, we could also have chosen non-blocking variants, which return an error code if the data is not ready, or if there is insufficient buffering. SW is always free to take advantage of the error codes returned by the interface methods to do other useful work. A reactive programming paradigm is often useful to improve application throughput using these error codes. All these techniques can help to overlap the communication latency with the computation, effectively improving the performance ❼.

## 3. A DIFFERENT PROGRAMMING MODEL

In this section, we examine the possibility of automating the steps detailed in Section 2 and to what extent our proposed programming model improves matters. A treatment of the related work is included in this discussion.

We require the high-level organization of the algorithm to be specified in terms of a data-flow graph [1]. In the data-flow model, nodes in the graph encapsulate computation, while directed edges represent communication. In order to execute the functionality at a node, data must be present on the input edges and buffering must be available on the output edges. The edges themselves are implemented as *guarded* FIFO channels, which means that they respect the data-flow contract and naturally enforce flow control. Figure 2.a shows an example of an application which has been decomposed into a data-flow graph consisting of six nodes. The edges in the graph reflect the underlying communication between application components, and the coloring indicates the partitioning choice (nodes colored blue have been selected for HW implementation). Rather than interlinked execution with a blocking RPC, interaction with the HW accelerator now involves enqueuing data into output FIFOs and dequeuing data from input FIFOs.

There is a substantial body of work surrounding algorithmic approaches to automating the choice of a HW/SW partitioning that will improve application throughput or energy efficiency. One such example is [2]. The success of these approaches depend on how accurately the *dynamic* properties of the application are captured in the models constructed by the programmer, as well as the cost model of the target platform. We have chosen not to automate the kernel selection, though any efforts in this area are complementary to the work presented in this paper.

It is important to note that the data-flow edges effectively isolate the functionality encapsulated by the nodes. When migrating functionality from SW to HW, it may need to be reimplemented in a different language, e.g. from C++ to Verilog, but as long as the functionality is selected at the same granularity as the nodes in the graph, this migration will not affect adjacent nodes. This isolation allows the designer to lower the reimplementation cost when experimenting with different HW/SW partitions.

Much research has been undertaken in the generation of specialized HW with varying degrees of automation. For applications with regular data-flow, a C-to-gates solution such as [3, 4] can generate an implementation from the original application SW. Cases where the data-flow is less regular or where the designer wants greater control over the micro-architecture, re-implementation in some HDL is required [5]. In contrast with C-to-gates solutions, the use of languages designed specifically for HW/SW codesign [6, 7] can fully automate the generation of interfaces and implementations, often with superior results. The Chinook [8] compiler is another tool interface generation.

As shown in Figure 2.b, the HW/SW interface is simply the union of all data-flow edges which cross the HW/SW boundary. We use Bluespec [9] to specify the node interfaces, since its guarded interface semantics provide a direct implementation of the data-flow contract. Below we give the Bluespec interface definition for the IFFT module discussed in Section 2:

```
typedef FixedPoint#(8,24) FxPt;
typedef Vector#(MAX_IFFT_SZ,FxPt) Frame;
interface IFFT
  method Action dataInput(Frame x);
  method ActionValue#(Frame) dataOutput();
  method Action frameSz(Bit#(3) x);  endinterface;
```

We assign one interface method to each data-flow edge. From this declaration, our interface compiler can easily infer the FIFO channels connecting HW and SW, the conversion routines for all data-types being transmitted, and the correct abstraction to expose to the SW application code.

In [7], the Bluespec Codesign Language (BCL) was used to define the entire data-flow graph, including the functionality at each node. This can be a convenient option when building an applicaiton from whole-cloth, but it can also be burdensome since it requires implementing the entire algorithm in BCL. In contrast, the methodology described in this paper provides an incremental approach which allows the programmer to exploit existing IP (specified in C++, Verilog, or VHDL) inside a data-flow graph whose structure is specified using Bluespec. Because data-flow graphs are hierarchically composable, any node can itself describe a subgraph specified entirely in BCL whose partitioning can be further explored. In further contrast with [7] which gave no details on the communication infrastructure, this paper provides a generic procedure for automatically synthesizing the HW/SW interfaces on any system for which "platform support" has been implemented.

Figure 2.b shows the data-flow graph in a state where the nodes have been grouped to form the HW and SW applica-

tion components. The task of connecting these components is simply a matter of implementing the FIFO channels on the shared communication fabric, and presenting a usable abstraction to the application-level HW and SW. Due to the latency tolerance of data-flow edges, we have some flexibility which can be exploited in the interest of efficiency.

The first step in implementing the FIFOs used in the programming model is to map them to the shared communication fabric. This synthesis problem is illustrated in Figure 2.c. Given a model of the fabric, marshaling and demarshaling routines must be generated for each data-type being transmitted. A scheduler must also be generated to ensure that all virtual FIFOs are fairly serviced. The scheduling logic is especially important when multiple CPU core are concurrently accessing different interface FIFOs over the same bus. So as not to introduce deadlocks, additional buffering may also need to be place in either HW or SW domains. To simplify the logic, we assume a static priority among the channels, though it is easy to imagine a more complex scheduler which reacts to the environment dynamically. If the user does not specify otherwise, we treat all FIFOs with the same priority and implement a simple round-robin schedule. Once an abstract model of the physical network has been manually specified, this synthesis problem can be automated completely.

The final step requires the implementation of the scheduler and marshaling/demarshaling routines on a physical platform. An interface compiler can generate low-level SW driver code and HW shims along the lines of our discussion in Section 2. The exact details of the generated code depend on the target platform.

### 3.1. Implementation Details

The methodology has been discussed in some abstraction, but we have implemented a working compiler which targets the Zedboard and ML507 platforms from Xilinx. Code generated by our interface compiler relies on platform support, which consists of some HW and SW libraries as well as an abstract model of the interconnection fabric for use by the compiler. To use our interface compiler, the user simply defines the guarded interface for the specialized HW in Bluespec, using only the unary *Action* and nullary *ActionValue* methods. On the SW side, the interface compiler generates a header file with the interface FIFO declarations, their C++ implementations, and the necessary driver code to communicate with the physical network. On the HW side, the compiler generates a communication scheduler and a shim for each FIFO channel. Shims consist of a slave interface which connects to the bus, an FSL master interface used to communicate with the scheduler, and additional buffering implemented using BRAMs. The scheduler implements one FSL slave interface for each FIFO channel, and a bus mastering interface for the initiation of bus transactions.

Figure 2.d shows how all the pieces fit together on the Zynq platform. When a SW thread wants to send data to the HW over a particular channel, it writes the data to the corresponding output FIFO implemented in system memory. If the FIFO lacks sufficient free buffering, an error code will be returned which the application SW must handle appropriately. After successfully writing the data to the SW FIFO, the driver notifies the communication scheduler of a pending transmission. The scheduling logic is a bus mastering device, which at some point will initiate a DMA transfer from system memory to the slave interface corresponding to that particular FIFO on the HW side. Transmission in the reverse direction takes place through similar means, but is initiated by the application HW instead of the SW, over one of the FSL links. Instead of communicating with the scheduler over the AXI bus, the FSL links provide a side-channel thereby reducing bus traffic.

It is easy to imagine a system with a more interesting communication fabric. The ML507, for example, has four dedicated DMA engines implemented in ASIC, as well as the Xilinx PLB standard bus. With such a network, we map the higher-bandwidth FIFOs to the DMA engines, since their dedicated interfaces allow for extremely efficient data transfer. FIFOs with lower bandwidth requirements might be implemented on the PLB. In general, we will need to synthesize one scheduler for each multiplexed physical connection in the interconnection fabric, so a more complex system may have more than one communication scheduler. Automating this process simply requires a more complex model to represent the communication fabric. Once the data-flow edges have been assigned to a particular physical link, the infrastructure supporting each link can be compiled separately in the manner outlined previously. The schedulers are aware of all FIFO interfaces and their relative priorities on a particular physical link. This provides a natural extension for the case where threads running on different cores are each attempting to communicate with a different interface.

**Optimizations:** The system described on the Zynq platform requires one burst transaction for the transmission of each interface type. If the bit-representation of the interface type is less than or equal to the width of the bus, the data could just as well have been written to that FIFO's slave interface directly. Depending on the burst setup time, there may be a range of bit-widths for which it may be more efficient to transmit the data directly even when more than one write is required. These tradeoffs are represented in the abstract model given to the compiler so it can synthesize the most efficient routines.

## 4. EVALUATION

To demonstrate our methodology, we employ it in a case study in which we attempt to identify a beneficial HW/SW partitioning choice for a number of applications running on the Zynq platform from Xilinx. We have selected what we believe is an interesting set of applications for embedded systems. In addition to the interface compiler, we have implemented a tool-chain that automates compilation, FPGA programming, and application execution. With this tool-chain we can automatically measure execution time, power consumption, and area (FPGA resource) costs for each partitioning choice. Time is measured by executing the application, power is estimated using XPA (Xilinx Power Analyzer), and FPGA resource consumption is directly reported in the place-and-route reports. Energy is computed on a per-component basis using the component's active time and the estimated power. In order to select the most suitable partitioning, a developer can combine these metrics at will.

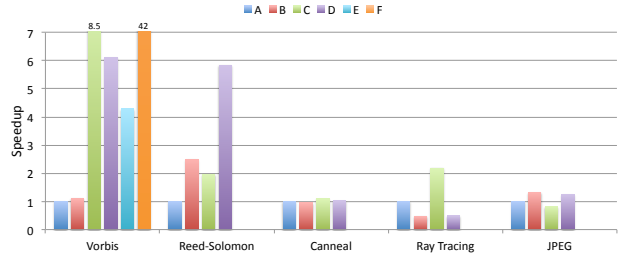



**Fig. 3**. Speedup obtained by the various HW/SW partitions over the full SW (A) partition (higher is better)

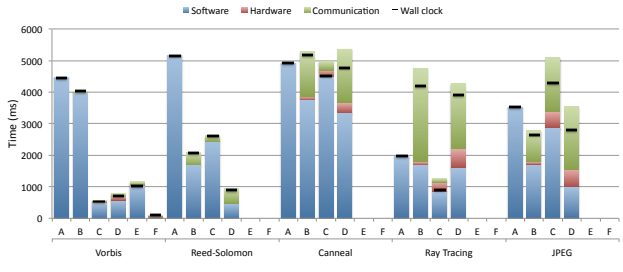



**Fig. 4**. SW, HW and communication times for each application partitions. The total execution time (a horizontal black bar) may be less than the sum of the three time components due to concurrent execution.

On the Zynq platform, HW implemented on the FPGA fabric has a much slower clock speed than the ARM core. Moving a component to HW will improve performance only if considerable HW parallelism can be exploited. For the cases where improved energy consumption is not accompanied by improved performance (a natural consequence of running for a shorter amount of time), it is because the HW reimplementation uses few FPGA resources.

What follows is an analysis of the benchmarks and an evaluation of the various partitioning choices. In Figure 3, we show, for all the benchmarks, the speedup obtained by each HW/SW partitioning over the full SW implementation (partition A). Vorbis has six partitionings (labeled A to F), while the other benchmarks have four (labeled A to D).

Figure 4 provides a breakdown of the execution time, for each partition, into the SW, HW and communication times. The total execution time is indicated by a black bar in each column. In Figure 5, we show for each benchmark, the energy consumption of the partitions, normalized to the full SW partition (A). Though not listed due to space considerations, FPGA-resource consumption ("area") must also be considered when multiple programs are simultaneously running on the same system, competing for finite resources.

**Ogg Vorbis Decoding:** Figure 6 shows the dataflow and the HW/SW partitions chosen for analysis. Moving the IFFT Core and Windowing function to HW, as in partitions C and F, results in a large speedup, seen in Figure 3. As seen in Figure 5, partition F achieves the least normalized energy consumption. The combination of the speedup and energy efficiency, coupled with a moderate area cost, make F an attractive partitioning choice. This is a result of the highly efficient and parallel HW implementation of the complex

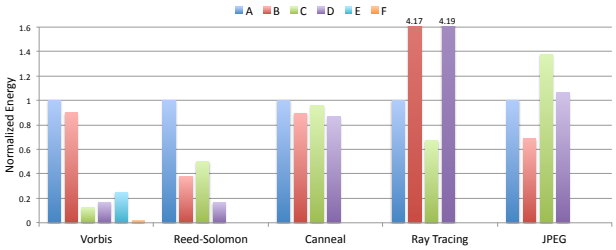



**Fig. 5**. Energy consumption of the HW/SW partitions normalized to the full SW (A) partition (lower is better)

and large IMDCT computation. Partition C presents an interesting alternative to F, since it achieves almost the same performance using far less FPGA resources.

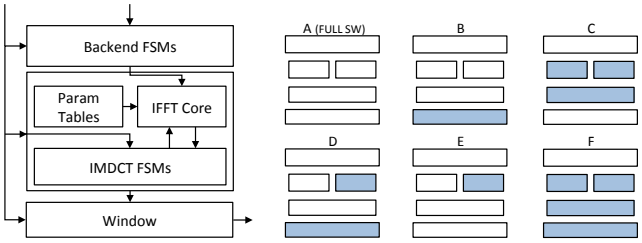



**Fig. 6**. Vorbis dataflow and partitions (HW is shaded)

**Reed-Solomon Decoding:** The Reed Solomon decoder is the error correction code used in the 802.16 receiver. Figure 7 shows the dataflow and HW/SW partitions chosen for analysis. Moving any block to HW leads to a large speedup due to the highly efficient parallel implementation of the Galois Field arithmetic blocks used in this algorithm. The active HW duration for all partitions of this benchmark is much smaller than the active SW and communication durations, as seen in Figure 4. Partition D has the maximum speedup (Figure 3) and the least normalized energy consumption (Figure 5) as it has all of the modules in HW.

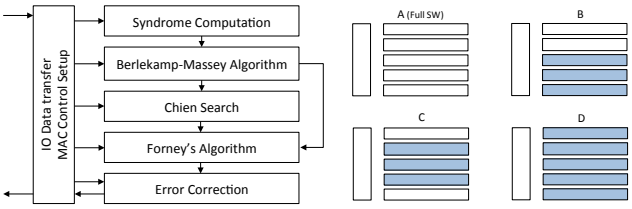



**Fig. 7**. Reed-Solomon partitions (HW is shaded)

**Canneal:** This member of the PARSEC benchmark suite performs simulated annealing to optimize the routing cost of chip design. The dataflow and partitioning choices are shown in Figure 8. This benchmark exhibits an interesting behavior. Partition C achieves the highest speedup over SW (Figure 3), but is not the most energy efficient. (Figure 5). This speedup happens when the communication latencies can be overlapped with the HW and SW computation, as shown in Figure 4. On the other hand, partitions B and D perform similar or slightly worse, but are more energy-efficient.

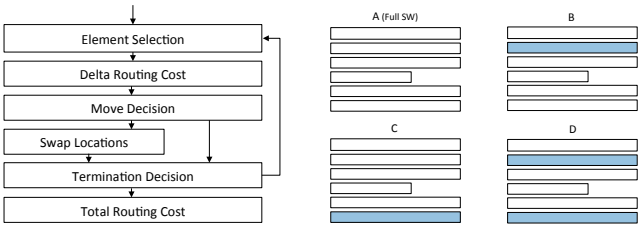



**Fig. 8**. Canneal partitions (HW is shaded)

**Ray Tracing:** This next benchmark is an implementation of ray tracing. Figure 9 shows the dataflow and the four partitions. Partition B moves the most intensive computations into HW, however this leads to the slowdown shown in Figure 3, since it generates pseudo-random memory accesses preventing efficient bursty communications. For Partition C, a single burst command transmits the necessary geometry to HW and the collisions are calculated by the pipelined HW, making it the most attractive choice. Attempting to put the traversal engine into HW, as in partition D, introduces similar synchronization hazards as in B.

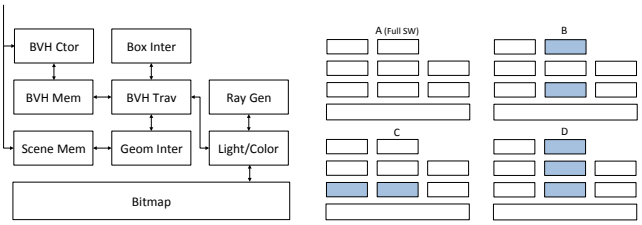



**Fig. 9**. Ray Tracing partitions (HW is shaded)

**JPEG Decoding:** This final benchmark implements the JPEG specification, whose data-flow and partitioning choices are shown in Figure 10. The results in Figure 3 show that only B achieves a speedup. HW/SW communication in partition B (2.2 MB) is smaller than in C (6.2 MB) and D (4.5 MB). This results in a smaller communication time (Figure 4) and reduced energy consumption (Figure 5). The reason that D does not show a greater improvement over A is because the performs the final image composition in SW.

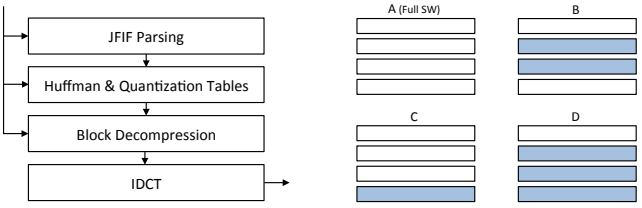



**Fig. 10**. JPEG Decoder partitions (HW is shaded)

**Final remarks:** We examined different HW/SW partitioning choices for five well-known applications. Using our methodology and interface compiler, we were able to greatly reduce the reimplementation burden when experimenting with partitioning choices by automating the infrastructure generation. For some of the applications, there is a trade-off between performance and energy consumption. In others it was possible to find an optimal partition by both metrics.

## 5. CONCLUSION

In this paper, we have proposed a new methodology for introducing FPGA-acceleration to applications running in SW. Once the application is expressed as a data-flow graph, the programmer designates which nodes are to be accelerated on the FPGA. Our interface compiler can automatically generate the communication infrastructure on a number of supported platforms, greatly reducing the programming burden. In addition, the data-flow model provides other benefits to the programmer by isolating functionality, promoting modular refinement, and extending naturally to single-language Codesign solutions such as BCL.

We have shown how the FIFOs which constitute the HW/SW interface can be efficiently and automatically synthesized on the Zynq platform, and through the case study have demonstrated the flexibility of the programming model. Using an interface compiler we have implemented specifically for this task, a programmer can introduce HW acceleration to an application running on the microprocessor, and experiment with multiple HW/SW partitioning choices without ever leaving the application space.

## 6. REFERENCES

[1] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *Int. Journal in Computer Simulation*, vol. 4, no. 2, 1994.

[3] *Catapult-C Manual and C/C++ style guide*, Mentor Graphics, 2004.

[4] Vivado Design Suite, `http://www.xilinx.com`.

[5] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed-solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72 –76, 2010.

[6] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," ser. ECOOP '08, 2008.

[7] M. King, N. Dave, and Arvind, "Automatic generation of hardware/software interfaces," ser. ASPLOS '12. New York, NY, USA: ACM, 2012.

[8] P. H. Chou, R. B. Ortega, and G. Borriello, "The chinook hardware/software co-synthesis system," in *In International Symposium on System Synthesis*, 1995, pp. 22–27.

[9] L. Augustsson, J. Schwarz, and R. S. Nikhil, "Bluespec Language definition," p. 95, 2001, Sandburst Corp.