2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Hardware/Software Communication Middleware for Data Adaptable Embedded Systems

Sachidanand Mahadevan, Vijay Shankar Gopinath, Roman Lysecky, Jonathan Sprinkle, Jerzy Rozenblit, Michael W. Marcellin
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ
{sachi, vsg, rlysecky, sprinkle, jr, marcellin}@ece.arizona.edu

***Abstract*—Recent trends toward increased flexibility and configurability in emerging applications present demanding challenges for implementing systems that incorporate such capabilities. The resulting application configuration space is generally much larger than any one hardware implementation can support. We present an overview of a new data-adaptive approach to rapid design and implementation of such highly configurable applications. In support of this data-adaptable approach, we demonstrate an efficient and flexible hardware/software communication middleware to support the seamless communication between hardware and software tasks at runtime. We highlight the flexibility of this interface and present an initial case study with results demonstrating the performance capabilities and area requirements.**

Keywords-Data adaptability; model-based design; hardware/software codesign; hardware/software communication middleware

I. INTRODUCTION

Significant increases in application complexity often demand processing requirements that exceed the performance achievable by current processors. At the same time, there is a trend toward increased flexibility and configurability in emerging applications that presents demanding challenges for implementing systems that incorporate such capabilities. While software implementations provide the flexibility needed to support standards – as demonstrated by the typical practice of releasing reference software code for these standards – the computational requirements of such applications often exceed the performance achievable by current processors. Alternatively, hardware accelerated solutions can potentially provide the required performance for these applications but are severely limited in the application space that can be supported.

JPEG2000 image coding standard precisely exhibits the aforementioned high degree of configurability and extreme computational demands. JPEG2000 offers flexibility at almost all stages of image compression including colorspace and bit-depth of the compressed image, options for wavelet transforms and quantization, support for both lossless and lossy compression, and Motion JPEG2000 for video support [15][28][33]. Many other emerging multimedia standards provide similar levels of configurability, including the MPEG-4 [13] and VC-1 [26] video encoding standards, among others.

In an effort to alleviate the design cost of developing software and hardware solutions capable of supporting the entire configuration space, many modern standards define several *profiles* intended for specific purposes. A *profile* either defines specific settings for various configurable parameters within the standard or a subset of allowed options, thereby reducing the level of complexity needed to implement a specific profile in software and/or hardware. While application profiles may allow hardware-accelerated implementations to support a subset of the application space, the number and variability of profiles even within a particular application domain precludes traditional hardware-based implementations as a viable option for many applications. For example, medical imaging applications are increasingly using JPEG2000 to meet stringent image quality requirements and to accommodate the necessary high compression to store and transmit vast amounts of data [16][23]. However, it is not the case that all medical imaging applications can use the same application profiles, or even that the same medical imaging hardware will always be used to visualize the same kinds of images.

Extensive research has demonstrated the benefits that can be obtained by hardware/software codesign and partitioning – with researchers and commercial vendors having achieved application speedups of 10X-1000X [2][7][9][11][21][22]. Hardware/software codesign provides a hybrid approach to design a system comprised of software running alongside custom hardware coprocessors or accelerators. However, a hardware/software implementation is still limited in the amount of configurability that can be supported in hardware due to area and cost constraints. Thus, a traditional hardware accelerated solution that supports all profiles even within a single domain is often infeasible, as hardware coprocessors must be generated for all profiles utilized within that domain. This is true for many configurable applications: a general hardware solution that covers the entire *data profile space*, or entire application standard, is infeasible since the combinatorial expansion of the design will not fit within a single affordable circuit.

Field-programmable gate arrays (FPGAs) – and reconfigurable computing in general – offer an increasingly economical alternative to traditional hardware-based solutions. Several FPGAs are currently available that integrate microprocessors and reconfigurable logic within the same integrated circuit. As such, hardware/software codesign targeting dynamically

978-0-7695-4379-6/11 $26.00 © 2011 IEEE
DOI 10.1109/ECBS.2011.12

reconfigurable FPGAs can be utilized to meet the performance demands and flexibility of modern multimedia applications. However, existing hardware/software codesign methodologies do not provide an efficient method to directly represent and exploit such *data configurability* at the application level. Instead, current approaches would require costly manual efforts to develop hardware for the various required data profiles.

In this paper, we present an overview of a new *data-adaptive* approach to enable the rapid design and implementation of highly configurable applications. In order to enable runtime reconfiguration and adaptive communication requirements of this approach, we present an efficient and flexible hardware/software communication middleware to support the seamless communication between hardware and software tasks at runtime. This communication middleware abstracts the communication and interfacing details required to efficiently transfer data between tasks, for which a task's implementation is not known a priori but rather will be determined at runtime. We highlight the flexibility of this interface and present an initial case study and results demonstrating the performance capabilities and area requirements.

We first provide a summary of related work in Section II. In Section III, we provide an overview of our data-adaptable methodology and demonstrate how this methodology can be utilized for highly configurable applications. In Section IV, we present a hardware/software communication middleware that provides seamless runtime communication between software and hardware tasks within our data-adaptable computing approach. In Section V, we present several experimental results highlighting the performance improvements that can be achieved using the presented communication middleware for JPEG image compression. Finally, we present our conclusions and provide an overview of future research direction in Section VI.

II. Related Work

Runtime reconfigurable and self-adaptive systems – extensively summarized in [2] and [29] – provide the opportunity to reconfigure an FPGA to implement hardware circuits for specific data profiles as needed by the target application given the current input data. In [31], a self-adaptive approach is presented that dynamically selects between software and hardware implementations for specific application kernels by profiling and monitoring function parameters. Alternatively, [30] proposes an adaptive design methodology that combines dynamic mapping of tasks between software and hardware alternatives with adaptive computing techniques that eliminates the need to re-execute computations that do not change for different application inputs.

Although these advancements are substantial, the design methods still produce restricted hardware implementations for specific application tasks and do not consider the entire application configurability. To support highly configurable applications, a design methodology is needed to capture and implement the entire application design and configuration space in order to produce suitable hardware and software partitions that are capable of meeting the required performance based on the expected data. Such a methodology, complete with new models of composition and system synthesis to inform existing tools and practices, would rapidly accelerate the availability of technologies currently in prototype to reach deployment.

Much research has also focused on developing methods for efficient communication or runtime management of hardware and software tasks implemented with reconfigurable systems, such as FPGAs. ReconOS [18][19][20] provides operating system support for scheduling and synchronizing hardware and software threads. Within this framework, each hardware threads is coupled with a software threads through which the OS can schedule the execution of hardware threads by scheduling the associated software thread. BORPH [25] provides a framework for supporting reconfigurable FPGAs within traditional operating systems by encapsulating hardware tasks in a custom executable format. This executable format provides both the configuration and interfacing details for hardware threads such that the operating system can manage the reconfiguration of the available FPGA resources. Within this framework, communication between software and hardware threads is supported through message passing protocols.

Complimentarily, by providing mechanisms for supporting POSIX synchronization constructs, e.g. mutexes, semaphores, POSIX threads can be implemented within both hardware and software [1]. Within this approach, distributed memory is utilized such that each hardware thread has a physically local memory that can be accessed by other software and hardware threads. Efficient communication can thereby be facilitated through pointers without out the need to copy data values.

Similar to our hardware/software communication middleware, several approaches have likewise utilized FIFOs to facilitate communication between software and hardware tasks. Both Williams et al. [34] and Xie et al. [35] propose utilizing tightly coupled FIFO-based communication, in which the processor has direct access to these FIFOs. For example, the Fast Simple Link (FSL) [32] supported within the Xilinx MicroBlaze processor provides an efficient communication link for fast communication between the microprocessor and directly connected components. In the former approach, software device drivers are utilized to provide access FIFOs through the FSL to efficiently communicate between hardware and software tasks. In the latter approach, a shared pool of FIFOs is utilized to communicate between multiple software tasks within multiprocessor systems,

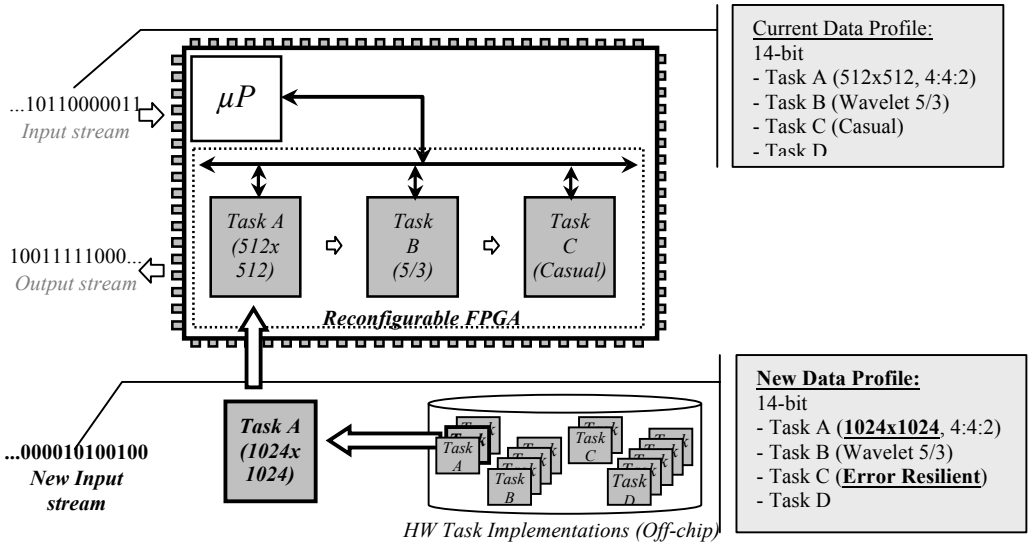

Figure 1. Hardware tasks are updated based upon the detected data profile of the incoming data stream. In this example, Task A will be reconfigured, as a hardware coprocessor is available for the newly detected tile size of 1024x1024. Hardware for Task C is no longer available and a software implementation will be utilized instead.

where the FSL links are utilized to access the shared FIFOs. For many applications, multiple tasks may compete for access to shared FIFOs. To support streaming applications exhibiting such multi-consumer, multi-producer access to shared buffers, Faure et al. [3] proposed a communication framework incorporating locking mechanism to enable both software and hardware tasks to access these shared buffers simultaneously.

III. Data-Adaptable Approach Overview

Figure 1 provides a conceptual overview of data-adaptable computing, presenting a particular configuration of software and hardware processing tasks along with alternative hardware task implementations that are available – though not currently in use – for an example streaming application. This example system consists of four tasks, *Task A-D*, where each task performs a different processing step of the overall application. For each processing task, hardware coprocessors may be available for a few specific data profiles that a designer identified as common data profiles needed within the target application domain, for which maximum performance is essential. The *data profile* of the incoming data provides and specifies the necessary information indicating which algorithms, or configurations of those algorithms, are needed to process the data correctly. For example, in most multimedia applications, the data profile of the incoming data is contained within the header of the incoming data stream or input file.

At runtime, the data profile of the current data input can be leveraged to determine if a suitable hardware implementation is available for any of the application's tasks, and will adapt the execution of the combined hardware/software implementation to use those coprocessors when possible. If a coprocessor is not available, a generic, non-accelerated software implementation can be utilized. Thus, data-adaptable computing seeks to maximize performance by adapting the execution at runtime by utilizing data profile specific hardware coprocessors, when available, required for the current input data being processed.

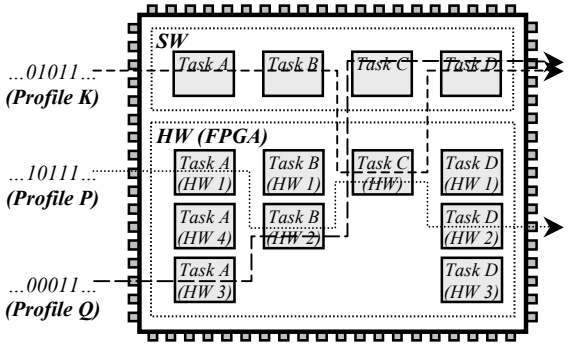

Figure 2. Overview of data-adaptability, in which the data profile of the incoming data stream determines which hardware tasks/coprocessors, if any, can be utilized to speedup performance.

For the example system presented, Figure 2 demonstrates the resulting software and hardware configurations for three different instances of incoming data that need to be processed, where the data profile for these distinct data inputs are indentified as *Profile K*, *Profile P*, and *Profile Q*. If the incoming data are identified with *Profile K*, only *Task C* can be accelerated using the hardware coprocessor, indicating that all other tasks will be performed in software. On the other hand, for *Profile P*, hardware coprocessors are available for all tasks within the applications, providing the best possible performance. Finally, in the case of *Profile Q*, although many hardware coprocessors are available for *Task D*, the

required coprocessor for this profile is different from the coprocessor needed for other profiles.

Implementing such a data-adaptable solution using traditional hardware based implementations is infeasible. Instead, FPGAs are perfectly suited for our data-adaptable approach as an FPGA can be dynamically reconfigured as needed based on the data profile of the incoming data.

Figure 1 further provides an overview of the data-adaptable hardware/software based implementation, targeting a system-on-a-chip (SOC) integrating a microprocessor and FPGA performing a JPEG2000 decompression/compression of an image stream. An initial input stream summarized as a 14-bit data profile is shown in its configuration. In this profile, *Task A* operates on 512x512 tiles of data, *Task B* takes the output of this task and performs a 5/3 wavelet transformation, and *Task C* applies a compression scheme to the output data stream. At some point, the input stream changes, and a new data profile is recognized. This new data stream uses 1024x1024 tiles for the first task, and the various other tasks – although not specified for a particular tile size – may also need to operate on the new 1024x1024 tiles. The existing hardware profile may not be suitable for the new tile size. Alternatives for computation of the new data stream are a new hardware implementation, if it exists, or the base software implementation. Since the existing hardware implementation is no longer needed, the reconfigurable hardware upon which it is running can be freed for use by hardware implementations of the new tile size.

By detecting the data profile of the incoming data stream, the data-adaptable approach can reconfigure the hardware implemented within the FPGA to best match the newly detected data profile. As previously mentioned, the data stream will normally contain all necessary information regarding the configuration of necessary computational elements. If a hardware coprocessor is not available for a specific task, the base configurable software implementation will be utilized for those tasks. As such, the approach is data-adaptable in that the system is able to adapt its execution based upon the data characteristics of the incoming data.

Importantly, it is the complexity of highly configurable applications that prevents the system designer from developing this data/profile reconfiguration strategy by hand. Given the combinatorial complexity of highly configurable applications, a designer would likely resort to ad hoc methods to cover the entire data profile space of interest. Our objective is to automate this profile space coverage through the specification and modeling of the configurable data and design parameters, and allow synthesis of hardware/software configurations that have the strength of hardware point solutions, with the flexibility to reconfigure themselves for optimal performance based on the incoming data stream.

To facilitate a hierarchical description of the target application as a set of independent communicating and configurable tasks, we have developed a new modeling technique termed communicating sequential dataflow tasks (CSDT). The CSDT model supports a rich set of semantics for specifying data profile generics and providing an execution model directly supporting the varying requirements imposed by data configurability. Our CSDT model – built using the Generic Modeling Environment [17] – supports a hierarchical framework for top-down implementation of a target application and its data configurability. This design environment is not tied to a particular language, but rather provides formalisms for the semantics visually tied to a graphical metamodel.

Within the CSDT model, tasks are independently modeled with communication specified through abstract

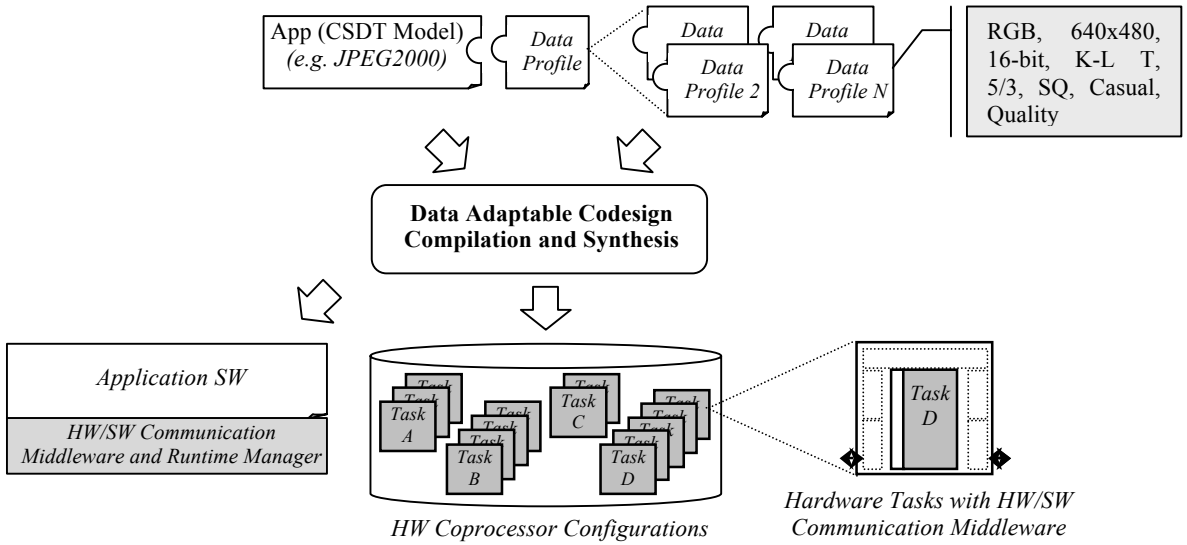

Figure 3: Data-adaptable hardware/software codesign methodology in which the data model compilation will create a minimal set of configured hardware coprocessors and communication interfaces needed to support the designer-specified CSDT model and data profiles.

communication channels that provide a bounded data storage and synchronization mechanisms. By utilizing abstract models of communication, designers are not burdened with specifying exact requirements for every communication channel for all possible data configurability. Instead, the CSDT modeling framework provides mechanisms for specifying the relation between data profiles and communication requirements.

Figure 3 provides an overview of the proposed data-adaptable codesign methodology. The input to the methodology is the CSDT model of the target application and a set of designer-specified data profiles. The data profiles define the specific configured instances of the target application that must be supported by the reconfigurable implementation. Given the CSDT model and data profiles, the data-adaptable codesign methodology will generate a set of hardware tasks utilizing the hardware/software communication middleware to abstract all communication between tasks. Similarly, the software application will utilize the communication middleware for all communication between tasks.

IV. Hardware/Software Communication Middleware

Within the CSDT model, all communication is specified using specific communication channels between tasks using software bounder buffers or hardware FIFOs. For software task implementations, a basic set of APIs is provided to allow the task to read data from its input buffer or write data to its output buffer, in which each buffer is defined by a specific channel ID. However, the actual location of the input and output buffers may change at runtime. For example, communication between two software tasks will likely utilize bounded buffers implemented in software and guarded with mutexes to control synchronization. In contrast, communication between two hardware tasks will use a hardware FIFO providing true simultaneous read and write access.

To abstract these communication details for the software and hardware task implementations, the hardware/software communication middleware utilizes a runtime manager to direct and synchronize data transfer to and from the appropriate software or hardware task implementation. This runtime decision is determined by both the data profile of the incoming data along with the availability of a hardware task required for the current data profile. To support these dynamic communication requirements, a communication middleware spanning across software and hardware boundaries is required. Given the target channel ID and the currently available hardware tasks, the runtime communication manager will automatically determine which software buffer or hardware FIFO is needed for the current read or write operation. For software-to-software transfers, standard guarded buffers are utilized, for which the runtime manager simply maintains a mapping of channel ID to specific software buffers.

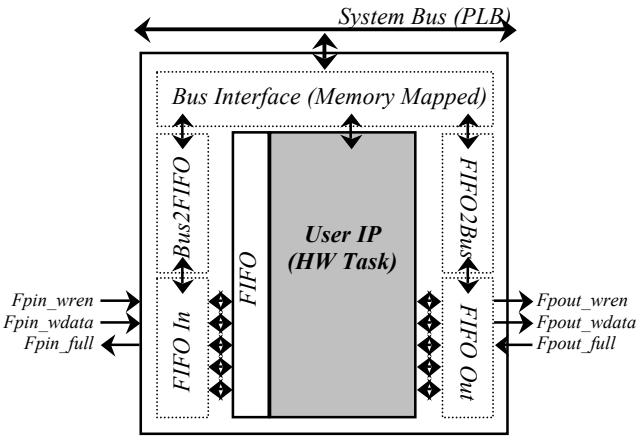

Figure 4. Hardware/software communication framework consisting of several abstracted communication components for hardware IP cores.

Alternatively, for software-to-hardware, hardware-to-hardware, and hardware-to-software communication, the runtime manager must also interface with and configure each hardware task for the current communication requirements defined by the input data profile. One the primary components within the hardware/software communication middleware is a hardware/software communication framework facilitating seamless runtime communication between software and hardware tasks suitable. Figure 4 presents an overview of the hardware/software communication framework for hardware tasks consisting of several abstracted communication components.

This communication framework supports both memory-mapped communications over the system bus and streaming data interfaces between tasks implemented in hardware. The flexibility provided by this communication framework allows application designers to choose the best method of communication to achieve optimal performance for the target application. This framework further provides support for streaming data communication between adjacent and non-adjacent hardware tasks implemented within an FPGA. As such, the common interface utilized by all hardware task implementations provides a clear delineation between individual hardware tasks while allowing hardware tasks to be placed within any reconfigurable region of the target FPGA – thereby providing near seamless support for dynamic reconfiguration.

The hardware/software communication middleware consists of seven components including: 1) the main IP core *(User IP)* implementing the hardware tasks desired functionality, 2) a FIFO for providing data input to the hardware task, 3) a streaming interface to receive incoming data from hardware or software tasks (*FIFO In*), 4) a streaming interface for writing data to an adjacent hardware task's FIFO (*FIFO Out*), 5) memory-mapped interface for directly accessing the system bus (*Bus Interface*), 6) an interface for providing memory-

mapped access to the FIFO (*Bus2FIFO*), and 7) an interface for using burst bus-based communication to non-adjacent hardware tasks (*FIFO2Bus*).

The *User IP* component defines the specific computation required for each tasks implemented within hardware. Within the DARES framework, the *User IP* supports the same computation – or subset of computations – as that supported by the original software tasks. Again, within the reconfigurable framework, the profile on incoming data will guide whether specific tasks will be executed in software or hardware at runtime. The *User IP* component can be created either manually by hardware designers or by utilizing high-level synthesis tools [3][4][10][12]. For all of the *User IP* cores considered within this paper, we utilized the ImpulseC CoDeveloper high-level synthesis tool [12] to automate the creation of hardware tasks from the original software code.

To support the communication abstraction, the input data for all hardware tasks is stored within the task's FIFO, which serves as the primary method for supporting the stream based communication between software and hardware tasks. The width of a task's FIFO is currently constrained to the width of the system bus. For the target systems considered within this paper, the system bus and FIFO width is limited to 32 bits. The depth of a task's FIFO must be at least as large as the size of data that needs to be processed during each execution of the hardware task.

Using the Xilinx IP Core Generator, a properly sized FIFO can be created for each hardware task. The specific FIFO implementation used within the hardware/software communication middleware requires independent synchronous read and write ports, a first word fall through to minimize communication latency, along with status signals indicating the current number of data items in the FIFO, if the FIFO is empty, and if the FIFO is full.

The *FIFOIn* and *FIFOOut* components provide streaming interfaces for transferring data between connected hardware tasks within the FPGA. This provides the fastest method of communication between tasks,

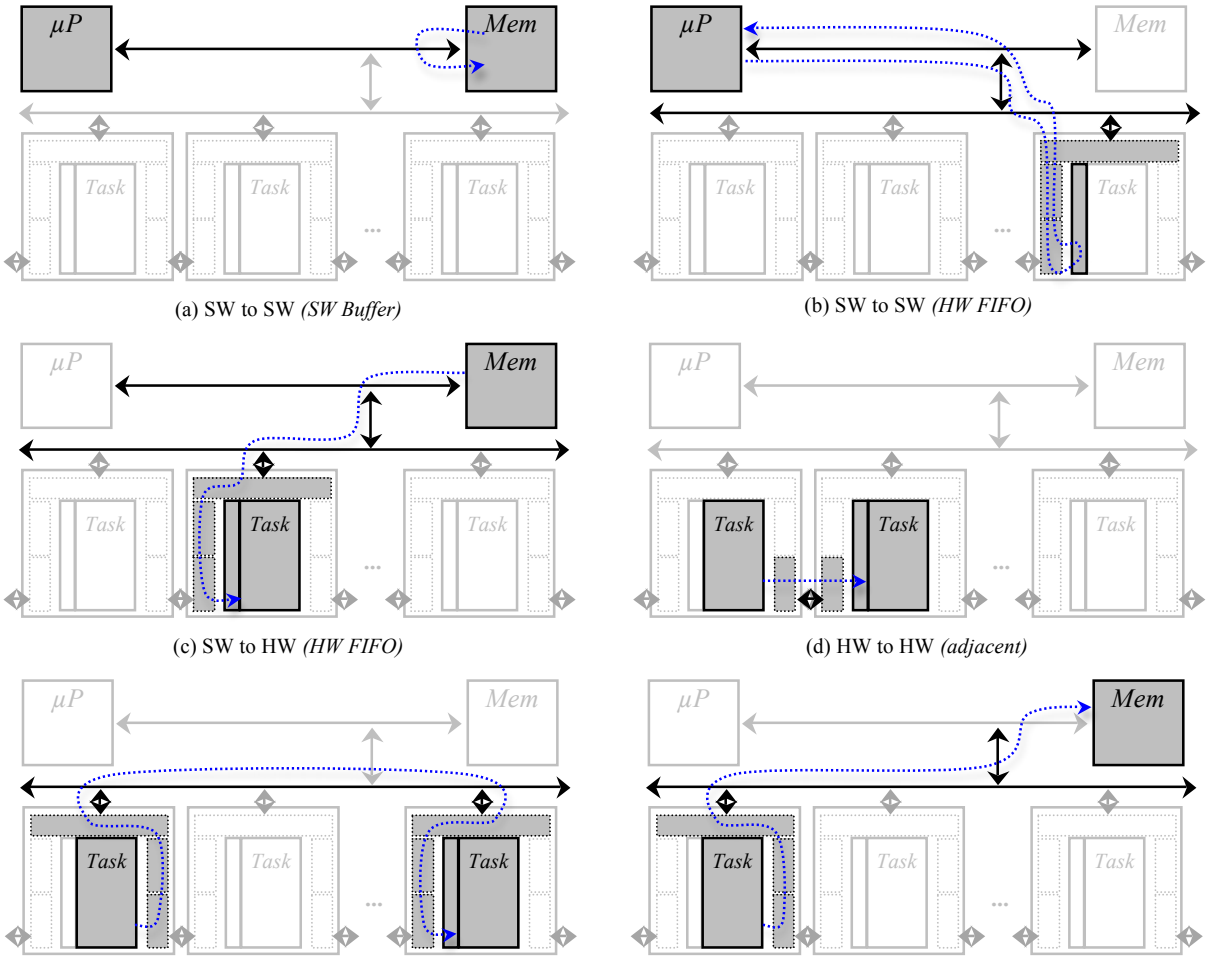

Figure 5. Hardware/software communication scenarios for software and hardware tasks: (a) software task to software task using memory based FIFO, (b) software task to software task using hardware FIFO, (c) software task to hardware task, (d) adjacent hardware task to hardware task, (e) non-adjacent hardware task to hardware task, and (f) hardware task to software task using memory based FIFO.

achieving a maximum throughput of one cycle per data transfer – assuming the sending tasks can provide data at this rate and the receiving task's FIFO is not full.

Additionally, access to a task's FIFO is also provided through a set of memory-mapped registers using the *Bus Interface* and *Bus2FIFO* components. These interfaces have been designed to support both single data read and write operations as well as burst DMA operations. Specifically, the *Bus2FIFO* interface is utilized to synchronize write access to the task's FIFO given possible contention for writing to the FIFO through the memory-mapped registers and *FIFOIn* interfaces. We note that although no contention between the *Bus2FIFO* and *FIFOIn* will occur within the DARES methodology, the current framework manages any contention by giving priority to the *FIFOIn* interface. In addition, the *Bus2FIFO* component synchronizes FIFO read and write accesses during DMA transfers to achieve a maximum throughput of one access per cycle as long as the DMA transfer can provide data fast enough and the task's FIFO is not full.

Output from the current task is transmitted to the *FIFOOut* interface. The *FIFOOut* interface will determine if the data needs to be written to the adjacent connected hardware task by directly writing the data to the *FIFOIn* interface of the adjacent hardware task or to a memory-mapped location for non-adjacent hardware and software tasks. For adjacently connected hardware tasks, the interfaces provided by the *FIFOIn* and *FIFOOut* utilize a subset of signals available from the receiving task's FIFO, including write control, write data, and FIFO full signals. For non-adjacent communication, the *Bus2FIFO* component utilizes burst data transfer over the system bus to transfer data to non-adjacent hardware tasks or software buffers in memory for software tasks.

The hardware/software communication middleware provides bus-based communication between hardware tasks and other systems components for non-streaming communication. The *User IP* can directly access the system bus using the *Bus Interface* component. This interface can be particularly useful for configuring and initializing hardware cores, as well as providing support for interfacing with system inputs and outputs.

The communication middleware was designed to support seamless communication between software and hardware tasks. Figure 5 provides an overview of supported communication methods between tasks, including:

- *SW to SW (SW Buffer)*: software task to software task using memory based buffer
- *SW to SW (HW FIFO)*: software task to software task using memory-mapped access to hardware task's FIFO
- *SW to HW (HW FIFO)*: software task to hardware task using hardware task's FIFO
- *HW to HW (adjacent)*: hardware task to hardware task using streaming *FIFOIn* and *FIFOOut* interfaces for adjacently connected hardware tasks

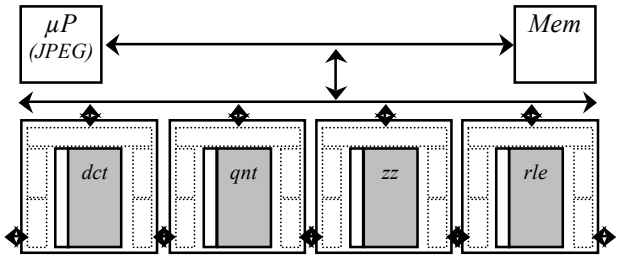

Figure 6. Hardware core implementation for JPEG encoding application.

- *HW to HW (non-adjacent)*: hardware task to hardware task using burst data transfers for non-adjacent hardware tasks
- *HW to SW (SW Buffer)*: hardware task to software task using DMA burst transfer to memory based buffer

V. Experimental Results

Although our intended goal is to apply this framework to support data adaptability for JPEG2000 image compression and decompression, we currently focus our initial development efforts on the less complex JPEG standard. Specifically, we consider a multithreaded software implementation for JPEG image compression. The initial multithreaded software implementation utilizes six POSIX threads for file input, discrete cosine transform (*dct*), quantization (*qnt*), zig-zag ordering (*zz*), run-length encoding (*rle*), and output encoding and file output. Communication between software tasks are implemented using software bounded buffers, to which access is guarded by mutexes. The initial multithreaded software application was implemented on a Xilinx ML507 development board incorporating a Virtex-5 FX FPGA. The software was executed on the PowerPC processor of the FPGA at the maximum operating frequency of 400 MHz with a processor local bus (PLB) frequency of 100 MHz. All software tasks were executed using the Xilinx XilKernel 4.0.

Within the JPEG encoding application, four of the software tasks, *dct*, *qnt*, *zz*, and *rle*, were identified and implemented as hardware tasks using the presented hardware/software communication middleware. The *User IP* component for each of these hardware tasks was directly created from the original software code using the ImpulseC CoDeveloper high-level synthesis tool. We note that the interface utilized by ImpulseC does not directly connect with the hardware/software communication framework. Hence, additional logic was utilized to interface and synchronize the ImpulseC generated cores with our communication framework. Figure 6 provides an overview of the resulting hardware/software implementation for the JPEG encoder in which communication between software and hardware tasks is managed by our communication middleware.

TABLE I. AREA REQUIREMENTS REPORTED IN LOOK-UP TABLES (LUTS) AND FLIP-FLOPS (FFS) FOR THE HARDWARE COMMUNICATION MIDDLEWARE FRAMEWORK (HWCM) AND FOUR HARDWARE TASK IMPLEMENTATIONS FOR THE JPEG ENCODING APPLICATION.

AREA	HWCM	HARDWARE TASKS				
		DCT	*QNT*	*ZZ*	*RLE*	***TOTAL***
LUTs	843	4811	2603	3037	4850	***19,693***
FFs	806	1895	2145	2247	2226	***13,866***
% HWCM		24%	35%	31%	23%	***19%***

Table I presents the area requirements reported in look-up tables (LUTs) and flip-flops (FFs) for the hardware communication middleware framework (HWCM) and each of the four hardware task implementations for the JPEG encoding application. The hardware communication middleware framework requires a total of 1649 LUTs and FFs, of which the majority of the required logic is due o the FIFO component needed to store data locally. For each of the four hardware tasks, the hardware communication framework correspond to between 23% and 35% of the total logic required for the tasks hardware circuit. For the quantization tasks, the large percentage required for the communication framework is primarily due to the simplicity of the quantization operation. Thus, the *User IP* component for this hardware task only requires 65% of the total logic for that task. However, for more complex operations, such as run-length encoding, the hardware required for the *User IP* core can be as much as 77% of the total area required. Overall the communication framework requires only 19% of the total logic resources required for all hardware tasks.

Several factors can affect the overall performance of the resulting hardware/software implementations. First, within the data-adaptable approach, only those kernels that match the current data profile can be accelerated using the hardware task implementations. We evaluated the performance of all possible hardware tasks combinations to analyze the potential performance benefits of the approach.

Second, for many applications the communication of data between tasks – especially between hardware and software tasks – limits the overall speedup that can be achieved. For these applications, a higher system bus frequency may help to improvement the overall performance by reducing the communication latency. At the same time, as the hardware tasks operate at the same speed as the system bus, the performance the hardware tasks can be further improved. However, using the Xilinx Virtex-5 FX FPGA, the ratio of the processor to clock frequency is restricted to specific ratios. Increasing the system bus frequency incurs a tradeoff of reduced processor frequency. Therefore, we consider two alternative processor to bus frequency ratios, including a 4:1 ratio in which the processor executes at 400 MHz and the system bus executes at 100 MHz and a 2:1 ratio in which the processor executes at only 250 MHz with the system bus operating at 125 MHz.

Lastly, we consider the impact of the using direct memory access (DMA) for transferring data between software and hardware tasks. Using the processor local bus within the target systems, individual bus transaction can incur wait times of up to 36 cycles. Transferring data one word at a time between tasks can significantly limit the overall throughout. For the JPEG encoding application, most of the hardware tasks operate on well defined blocks of data that allow data to be transferred using burst operations via DMA. To evaluate the performance benefits with and without DMA support, we consider three options for transferring data between tasks, including using individual write operation for each data transfer, using DMA to transfer one block of data between tasks, and using DMA to transfer four blocks of data between tasks during each transfer.

Figure 7 presents the normalized execution time for the JPEG encoding application for all combinations of the tasks implemented within hardware considering processor and system bus frequencies of (a) 400 MHz and 100 MHz and (b) 250 MHz and 125 MHz, respectively. All execution times are normalized to the original multithreaded software execution on the processor operating at the maximum frequency of 400 MHz. All

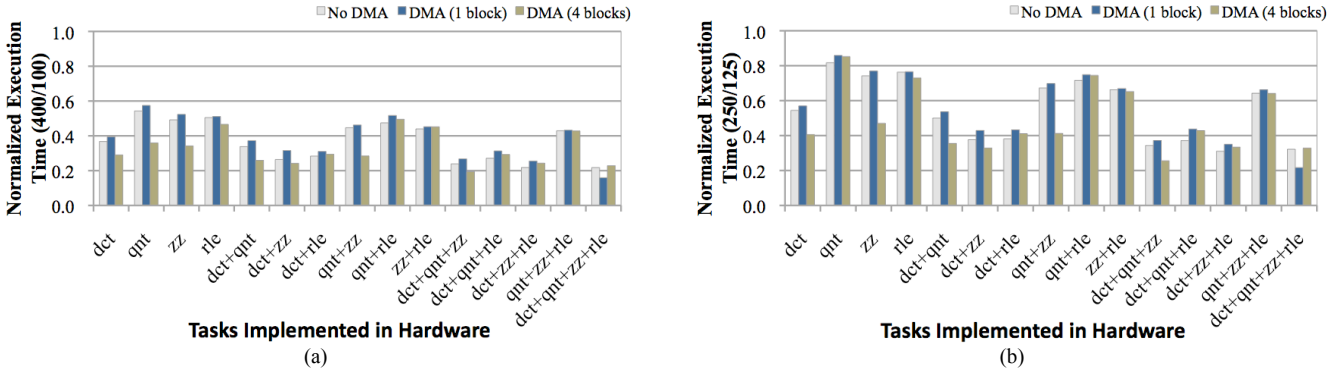

Figure 7. Normalized execution time for the JPEG encoding application for various combinations of the tasks implemented within hardware with processor and system bus frequencies of (a) 400 MHz and 100 MHz and (b) 250 MHz and 125 MHz. All execution times are normalized to the original software execution on the processor operating at the maximum frequency of 400 MHz.

execution times were determined using physical measurements from the Virtex-5 FX FPGA development board.

Overall, the highest performance is achieved by utilizing all hardware tasks (*dct+wnt+zz+rle*) using operating frequencies of 400/100 MHz and DMA transfers of one block. This configuration yields a 6.3X speedup over the initial software application. Notably, for this set of hardware tasks, four block DMA transfers achieve lower performance. than using individual blocks for all but the fastest implementation. For all other implementations, transferring four blocks simultaneously results in increased performance. This difference in performance is primarily due to cache coherency and setup times involved within these DMA operations. As such a tradeoff exists between the reduced bus transaction wait times and the DMA setup and cache coherence times.

Although the 250/125 MHz implementation provides higher performance for data transfer and hardware task execution, these performance benefits are outweighed by the reduced processor frequency. For those tasks implemented in software, this slowdown significantly impacts the overall performance. While the best performing design for this configuration again utilizes all hardware tasks (*dct+wnt+zz+rle*), the overall speedup is only 4.6X – a 1.4X slowdown compared to the 400/100 MHz implementation.

We further examine the situation in which only a subset of hardware tasks are available for the current data profile. For scenarios in which the data profile only allows two hardware tasks to be utilized, performance speedups of 1.9X to 4.1X and 1.3X to 3.0X can be achieved for the 400/100 MHz and 250/125 MHz alternatives, respectively. For these scenarios, the size of DMA transfers can have significant performance impacts. For example, using the *qnt* and *zz* hardware tasks (*qnt+zz*), four block DMA transfers achieve a 1.6X speedup compared to the single block DMA transfers. Furthermore, for some implementations (*dct+rle*, *qnt+rle*, and *zz+rle*), individual data transfer provide higher performance compared to using DMA – as mush as 1.1X improvement. This performance improvement using individual data transfers is due to the variable size output produced by the *rle* task. Because the transfer sizes cannot be predicted in advance for *rle*, the latencies required reading this data using DMA transfer is increased, which affects the overall performance.

VI. Conclusions

The trend toward increased flexibility and configurability in emerging applications presents demanding challenges for implementing systems that incorporate such capabilities. For embedded applications, hardware solutions that reduce power consumption or increase speed may be infeasible if expected to cover the entire configuration space. This paper described a new approach to managing this complexity through a data-adaptable computing. We presented a hardware/software communication middleware that provides seamless support for runtime communication between hardware and software tasks and demonstrated that performance improvement of greater than 6X can be achieved with small additional logic resources.

While we have highlighted the potential benefits of the data-adaptable methodology and hardware/software communication middleware, important future work remains. Following efforts are focused on applying this technique to JPEG2000 image compression and decompression, which are both highly configurable and significantly more complex compared to JPEG compression. We are also developing a runtime reconfiguration framework to dynamically reconfigure the FPGA in response to changes in data profiles leveraging the hardware/software communication framework. Additionally, future work includes developing analytical models that can be utilized to determine the best communication method between tasks, without requiring long simulations or exhaustive prototyping efforts.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant CNS-0915010.

References

[1] Anderson, E., J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, D. Andrews. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. Symposium on Field-Programmable Custom Computing Machines, pp. 89-98, 2006.

[2] Banerjee, P., Mittal, G., Zaretsky, D., and Tang, X. BINACHIP-FPGA: A Tool to Map DSP Software Binaries and Assembly Programs onto FPGAs. In Proceedings of the Embedded Signal Processing Conference (GSPx), 2004.

[3] Buyukkurt, B., Z. Guo, W. A. Najjar. Impact of Loop Unrolling on Area, Throughput and Clock frequency in ROCCC: C to VHDL Compiler for FPGAs. International Workshop on Applied Reconfigurable Computing, 2006.

[4] C-to-Verilog, www.ctoverilog.com, 2010.

[5] Digital Imaging and Communications in Medicine (DICOM), National Electrical Manufacturers Association, 2008.

[6] Faure, E., A. Greiner, D. Genius. A Generic Hardware/Software Communication Mechanism for Multi-Processor System on Chip, Targeting Telecommunication Applications. Conference on Reconfigurable Communication-Centric SoCs (ReCoSoC), pp. 237-242, 2006.

[7] Gajski, D., F. Vahid, S. Narayan, J. Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System

Design. IEEE Transactions on VLSI Systems (TVLSI), Vol. 6, No. 1, pp. 84-100, 1998.

[8] Garcia, P., K. Compton, M. Schulte, E. Blem, W. Fu. An Overview of Reconfigurable Hardware in Embedded Systems. EURASIP Journal on Embedded Systems, Vol. 2006, No. 1, pp. 1-19, 2006.

[9] Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K. Optimized Generation of Data-Path from C Codes. In Proceedings of the Design Automation and Test in Europe Conference (DATE), pp. 112-117, 2005.

[10] Gupta, S., N. Dutt, R. Gupta, A. Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. International Conference on VLSI Design, pp. 461-466, 2003.

[11] Henkel, J., R. Ernst. A Hardware/Software Partitioner Using a Dynamically Determined Granularity. Design Automation Conference (DAC), pp. 691-696, 1997.

[12] Impulse Accelerated Technologies. Impulse CoDeveloper, www.impulseaccelerated.com, 2010.

[13] ISO/IEC 14496. Information technology -- Coding of Audio-Visual Objects, 2004.

[14] Jidin, R., D. Andrews, W. Peck, D. Chirpich, K. Stout, J. Gauch. Evaluation of the Hybrid Multithreading Programming Model using Image Processing Transforms. International Parallel and Distributed Processing Symposium (IPDPS), 2005.

[15] Joint Photographic Experts Group. JPEG2000 Image Compression Standard, www.jpeg.org/jpeg2000/.

[16] Krishnan, K., M. W. Marcellin, A. Bilgin, M. S. Nadar. Efficient Transmission of Compressed Data for Remote Volume Visualization. IEEE Transactions on Medical Imaging, Vol. 25, No. 9, pp. 1189-1199, 2006.

[17] Lédeczi, Á., Á. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. IEEE Computer, Vol. 34, No. 11, pp. 44-51, 2001.

[18] Lubbers, E., M. Platzner. Cooperative Multithreading in Dynamically Reconfigurable Systems. Field Programmable Logic and Applications (FPL), pp. 551-554, 2009.

[19] Lubbers, E., M. Platzner. A Portable Abstraction Layer for Hardware Threads. Field Programmable Logic and Applications (FPL), pp. 17-22, 2008.

[20] Lubbers, E., M. Platzner. ReconOS: An RTOS Supporting Hardware and Software Threads. Field Programmable Logic and Applications (FPL), pp. 441-446, 2007.

[21] Olson, J., J. Rozenblit, C. Talarico, W. Jacak. Hardware/Software Partitioning using Bayesian Belief Networks. IEEE Transactions on Systems, Man and Cybernetics, Vol. 37, No. 5, pp. 665-668, 2007.

[22] Rozenblit, J., K. Buchenrieder. Codesign: Computer-Aided Software/Hardware Engineering, IEEE Press, 1994.

[23] Schelkens, P., Munteanu, A., Barbarien, J., Galca, M., Giro-Nieto, X., Cornelis, J. Wavelet coding of volumetric medical datasets. IEEE Transactions on Medical Imaging, Vol. 22, No. 3, pp. 441-458, 2003.

[24] Schulz, S., J. Rozenblit, M. Mrva, K. Buchenrieder. Model-Based Codesign. IEEE Computer, Vol. 32, No. 8, pp. 60-68, 1998.

[25] So, H., R. Brodersen. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH. ACM Transactions on Embedded Computing Systems (TECS), Vol. 7, No. 2, Article 14, pp. 1-28, 2008.

[26] SMPTE 421M-2006, VC-1 Compressed Video Bitstream Format and Decoding Process, 2006.

[27] Stitt, G., F. Vahid, S. Nematbakhsh. Power Savings and Speedups from Partitioning Critical Loops to Hardware in Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Vol. 3, No. 1, pp. 218-232, 2004.

[28] Taubman, D., M. W. Marcellin, JPEG 2000: Image Compression Fundamentals, Standards and Practice, Kluwer International Series in Engineering and Computer Science, Secs 642, 2001.

[29] Santambogio, M. From Reconfigurable Architecture to Self-Adaptive Autonomic Systems. International Conference on Computational Science and Engineering, 2009.

[30] Santambogio, M., Memik, S., Rana, V., Acar, U., Sciuto, D. A Novel SOC Design Methodology Combining Adaptive Software and Reconfigurable Hardware. International Conference on Computer-Aided Design (ICCAD), 2007.

[31] Sima, V., Bertels, K. Runtime Decision of Hardware or Software Execution on a Heterogeneous Reconfigurable Platform. International Symposium on Parallel and Distributed Processing (IPDPS), 2009.

[32] Venkataramani, G., W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES), pp. 116-125, 2001.

[33] Special Section on JPEG 2000 Digital Imaging, IEEE Trans. on Consumer Electronics, Vol. 49, pp. 771-888, 2003.

[34] Williams, J., N. Bergmann, X. Xie. FIFO Communication Models in Operating Systems for Reconfigurable Computing. Field-Programmable Custom Computing Machines (FCCM), pp. 277-278, 2005.

[35] Xie, X., J. Williams, N. Bergmann. Asymmetric Multi-Processor Architecture for Reconfigurable System-on-Chip and Operating System Abstractions. Field-Programmable Technology (FPT), pp. 41-48, 2007.

[36] Xilinx, Inc. Fast Simplex Link (FSL), www.xilinx.com/products/ipcenter/FSL.htm, 2010.