The Hardware Thread Interface Design and Adaptation on Dynamically Reconfigurable SoC

Ying Wang, Wei-Nan Chen, Xiao-Wei Wang, Hong-Jun You, Cheng-Lian Peng
School of Computer Science and Technology
Fudan University
Shanghai, P.R.China
clpeng@fudan.edu.cn

Abstract— Nowadays, one of the challenges for creating a mixed hardware/software application on dynamically reconfigurable SoC is how to provide a unified programming model for hybrid hardware/software tasks and a portable interface adaptation for dynamically reconfigurable hardware tasks. In this paper, a POSIX-compliant hardware thread interface is proposed for data stream driven applications, serving for unified hardware/software multithread programming. At the same time, the stub/interface adaptation mechanism is also presented to support shared buffer based inter-thread communication/synchronization. At last, the experimental results on AES encryption/decryption hardware show that the interface design and adaptation could exploit programming transparency while effectively keep hardware efficiency.

Keywords—hardware thread; stub thread; interface adaptation

I. INTRODUCTION

With the development of reconfigurable logic devices, Dynamically Reconfigurable System-on-Chip (DRSoC), integrating dynamically reconfigurable hardware with embedded processor on the same die [1], is always used for data stream driven, mixed hardware/software applications like data encryption, signal processing, multimedia processing and so on [2][3][4]. The absence of a unified programming model for hybrid hardware/software tasks and a portable hardware task interface adaptation remains challenging to create an efficient application on DRSoC [5][6].

For hardware tasks implemented on reconfigurable hardware, the technology of dynamic reconfiguration makes it created and exit at run-time according to the demand of the application. It is possible for hardware tasks to be managed by the operating system like software tasks. The importance on providing a unified hardware/software task interface design and adaptation is well acknowledged by a lot of works [5][6][7]. According to the works of hthreads project [5], hardware task works analogously to general software thread and demands hardware implemented Operating System (OS) components for inter-task communication/synchronization. In contrast, our hardware task runs as a data processing hardware thread, which simplifies the Hardware Thread Interface (HTI) design and suitable for data stream driven application. Moreover, unlike the design reported in [6], which includes the platform-specific Window Management Unit (WMU) for virtual addresses translation, we investigate hardware thread interface adaptation in real mode for the sake of portability. On the other hand, the project of BORPH [7] presents hardware file I/O and IOREG interface to facilitate hardware process abstraction, allowing hardware process and software process to communicate through standard UNIX file pipes. In addition, the studies above have not considered the dynamic creation and execution of the hardware tasks.

Making use of our previous works on OS kernel extension and hardware tasks scheduling [8][9]. In this paper, we adopt unified hardware/software multithread programming model as our solution for hybrid hardware/software task design. For data stream driven applications executed on Xilinx 3.0, a standard POSIX-compliant HTI and a portable stub/interface adaptation are proposed to exploit programming transparency while keep hardware threads efficiency. The experimental results on data encryption/decryption application show that both the inter-thread communication/synchronization overhead and the resource utilization of the HTI are maintained at a low level.

This paper is organized as follows: next section will make a brief view on unified hardware/software multithread programming model. In Section 3, we will introduce the HTI implemented on bus based DRSoC. Section 4 illustrates the stub/interface adaptation built on HTI. As a proof of concept, the experiments on AES encryption/decryption thread are presented in Section 5. Finally, Section 6 concludes the paper.

II. UNIFIED MULTITHREAD PROGRAMMING MODEL ON DRSoC

Pthreads [10] is a widely used programming model in software development, which has been shown as a promising solution for unified thread abstraction for hybrid hardware/software tasks. The Unified Software/Hardware multithread programming Model supporting Dynamic Reconfiguration (SHUMDR) we proposed aims at hiding the implementation details of the hardware task and offering a unified hardware/software thread view [8].
Figure 1 depicts the layered structure of SHUMDR, which consists of four abstraction levels: hardware platform on DRSoC, HTI design and adaptation, extended OS kernel and Pthreads compatible SHUMDR, from low to high respectively. HTI design and adaptation plays an important role for providing unified hardware/software multithread programming model and it is necessary to give a deep and systematic research on HTI adaptation.

This paper will focus on HTI design and adaptation. Only under this design and adaptation, it is possible to provide a unified thread view on hybrid hardware/software tasks at higher level.

III. HARDWARE THREAD INTERFACE DESIGN AND ADAPTATION

Unified multithread programming model proposed here is designed for stream driven, mixed hardware/software applications. These applications compose a lot of computation intensive tasks demanding for simultaneous hardware acceleration, schedule flexibility and programming transparency. The concept of thread provides a suitable abstraction for these computation intensive hardware tasks.

According to the characteristics of the hardware tasks in data driven application, once a hardware thread is successfully created at run-time, its behavior can be divided into three phases: data reading, data processing and result writing. Used in this way, hardware tasks can be abstracted to concurrently running hardware threads managed by OS kernel. A portable hardware thread interface design and its adaptation should be provided for hardware thread abstraction.

A. Hardware Thread Interface Design

A hardware thread always comprises user defined hardware task (UHT) and hardware thread interface (HTI). UHT is used for stream driven computation, while HTI takes the characteristics of the stream-based computation into account and provides POSIX-compliant hardware thread abstraction. This structure makes it possible for OS kernel to manage the hardware task in the same way as the software thread.

During the process of hardware thread abstraction, in order to take advantage of the spatial locality for performance improvement, it is useful to include a local input/output memory within HTI.

As illustrated in Figure 2, upon our system bus based DRSoC, there are two parts inside the HTI: bus interface and task interface. Bus interface is responsible for address decoding and bus-macro [11] based signal control. At the same time, task interface responses to the communication and control primitive of OS kernel through state controller, parameter registers, command register and state register. The key of the task interface design is the state controller, which gives the possible state transition throughout the life cycle of the hardware thread according to the semantics of Posix thread.

As describes in Figure 3, we define three Posix thread compliant states: READY, RUN and WAIT. Hardware thread enters into READY state when successfully created or reset, it then turns into RUN state upon receiving cmd_run command. While at run state, if input FIFO is empty or output FIFO is full, it will immediate assert usr_data_transfer signal for input/output data transfer, blocking itself and entering into WAIT state. Hardware thread will return to RUN state upon receiving cmd_transfer_done signal. Additionally, hardware thread could go back to READY state when receiving cmd_stop command from OS kernel or user_stop signal within it.
Through thread interface design, a running hardware thread has access to local memory resource, reading data, performing computation and writing result. The hardware thread control and inter-thread communication/synchronization between software thread and hardware thread will be investigated in interface adaptation at high level.

B. Stub Thread based Interface Adaptation

In order to give programming transparency while keeping hardware efficiency, every running hardware thread has its stub thread accordingly. As a software representative of the hardware thread, stub thread is created synchronously with the hardware thread. It is used for redirecting the OS kernel control primitive to specific hardware thread and dealing with the communication/synchronization between software thread and hardware thread.

Once hardware thread is successfully created, its stub thread will dynamically bind the basic parameters. These parameters include hardware thread ID, thread state, base address, input/output FIFO address and depth, which will contribute to hardware thread control and inter-thread communication.

The inter-thread communication occurs on data transfer. Instead of putting a DMA engine inside every HTI for stream reading and writing as works of [6] does, we employ Central DMA Controller for data transfer for the sake of portability. The hardware thread only needs to have slave capability and the centralized DMA can be used as a memory-to-FIFO and FIFO-to-memory copy engine. The communication between software thread and hardware thread can be carried out by extended hardware thread API, which makes central DMA controller transfer data stream to and from local FIFO.

From low level state transition depicted in Figure 3, it can be seen that the communication between software thread and hardware thread is in a blocked way, which is suitable for producer-consumer application. At higher abstraction level, stub thread shares the same address space and resource with other software threads within a user defined process. Instead of using traditional Inter-Process Communication (IPC) or other complex communication mechanism, shared buffer based communication is an efficient way for hardware/software threads to share a large block of data. The existence of the stub thread could redirect the communication between hardware thread and software thread to be implemented by extended hardware thread API.

Figure 4 gives the working scenario of data stream driven producer-consumer application, where shared data buffer can be used for inter-thread communication through stub thread based interface adaptation. The thread has access to shared buffer through signal synchronization according to producer-consumer relationship.

There are two kinds of buffer allocation which will influence the data transfer overhead to some extend. The first uses the predefined on-chip BRAM as shared buffer for static buffer allocation; the second employs dynamic buffer allocation at run time and the allocated buffer is always located in SDRAM. Despite the different throughput on BRAM and SDRAM, the issue of cache coherence needs to be considered because of using DMA.

To take the scenario of Figure 4 as an example, if Shared_buffer_1 and Shared_buffer_2 are static allocated buffers in BRAM, because the address spaces of these two buffers are determinate at design time, we choose to mark these address spaces “un-cacheable” for eliminating the coherence operation and avoiding data cache pollution. For dynamically allocated buffers in SDRAM, as OS Kernel runs on this large size SDRAM, DCache should always be enabled. In order to keep cache coherence when SDRAM memory region uses write back caching policy, a cache flush is needed to update Shared_buffer_1 before copying source data stream to the hardware thread, a cache invalidate should be used before transferring result data stream to Shared_buffer_2.

Figure 5. The address space of the process with dynamic buffer allocation support

With the hardware thread interface adaptation, Figure 5 illustrates the address space of a process (described in Figure 4) when using dynamic buffer allocation. It can be seen that the stub thread stores the basic parameters in its stack segment, these parameters are used for actual hardware thread control and data transfer. The shared data buffer located in heap segment can be accessed by stub thread and other software threads, and these accesses could be...
synchronized by using synchronization mechanism defined in Pthreads library. The interaction between hardware thread and software thread will be largely simplified and contribute to provide programming transparency.

IV. CASE STUDY AND EXPERIMENTAL RESULTS

In order to validate the hardware thread interface adaptation, we have implemented data stream driven AES encryption/decryption application on our prototype described in Figure 6.

**Figure 6.** The architecture of our prototype

The prototype is built on XC2VP30 FPGA, where DCache and ICache are enabled. ICAP is responsible for dynamic hardware thread creation [12]. The PowerPC CPU, system bus, peripherals, external DDR memory, as well as hardware threads are all clocked at 100M Hz. Our previous works on Xilkernel 3.0 extension is also deployed on this prototype, where two hardware threads of AES encryption/decryption are time-sharing the reconfigurable logic resources. These two hardware threads can be created or exit at run-time according the requirement of the application.

Hardware tasks of data processing are abstracted to POSIX-compliant hardware threads under the support of hardware thread interface design and adaptation. The resource consumption of HTI is kept at a low level, accounting for 3.2%, 2.8% and 2% of the total number of Slices, Flip Flops and LUTs on XC2VP30 respectively.

In our stream driven encryption/decryption application, data to be processed is stored in Compact Flash card in binary file format. At first, producer thread writes the data block to shared input buffer. Then data encryption/decryption hardware thread reads the data block, does computation and writes the results to output buffer. Finally, consumer thread reads the result data from shared output buffer for display. Through HTI adaptation with dynamically allocated buffer support, the application can be described from a unified hardware/software multithreading view as illustrated in Figure 7.

**Figure 7.** Stream driven encryption/decryption application with a unified hardware/software multithread view

As described in Section 3.2, the way of shared buffer allocation implies different method on cache coherence, thus making the inter-thread communication time different. It is assumed that the local memory size within HTI is equal to the size of shared buffer. The results of using dynamic or static allocation with different buffer size are listed in Table 1.

**TABLE I.** Communication Overhead when Using IA with Dynamic Buffer Allocation Support

<table>
<thead>
<tr>
<th>Shared Buffer Size (Bytes)</th>
<th>Communication Overhead from SW Thread to HW Thread (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IA with Dynamic Buffer Allocation Support</td>
</tr>
<tr>
<td></td>
<td>DCache Flush</td>
</tr>
<tr>
<td>2K</td>
<td>85.64</td>
</tr>
<tr>
<td>4K</td>
<td>167.56</td>
</tr>
<tr>
<td>8K</td>
<td>331.41</td>
</tr>
</tbody>
</table>

It can be seen that although Interface Adaptation (IA) with dynamic buffer allocation support spends more time in avoiding data pollution and results in large communication overhead, it brings more programming transparency and flexibility. Despite of using a little more time when transferring the same size of data because of un-cached BRAM, IA with static buffer allocation support is efficient for it is free of data pollution.

According to our interface adaptation, when hardware thread needs to access the shared buffer for data reading/writing, it always turns to its stub thread through hardware thread interface design given in Section 3.1. Stub
thread behaves on behalf of the hardware thread and implements inter-thread synchronization through traditional synchronization primitive. We measure the time for one waiting thread to acquire the semaphore released by another. Figure 8 shows the experimental results under Xilkernel 3.0.

For synchronization from software thread to hardware thread (ST-HT), a waiting hardware thread will be continued upon receiving semaphore posting notification from its stub thread. It takes additional time on semaphore notification than synchronization between two software threads (ST-ST). For synchronization request initiated by hardware thread, the request for semaphore is sent to its stub thread via interrupt handler, bringing long interrupt latency than software thread initiated synchronization. It should be noted that although communication/synchronization between two hardware threads is supported here, the stub thread based adaptation is suitable for applications with little or no communication between hardware threads.

In our data encryption/decryption application running on Xilkernel 3.0, we choose to design hardware thread with the hardware interface design and adaptation support, where shared buffer size and local FIFO size are all set to 2K Bytes. The hardware thread creation is based on dynamic reconfiguration and tens of milliseconds are spent on hardware thread creation at run-time. The performances of different types of thread on data encryption/decryption are compared in Figure 9 (a) and (b).

Before different types of thread starts to work, the time used for software thread creation is almost 19 us, while the time spent on encryption/decryption hardware thread creation is 32.3 ms and 31.54 ms, respectively. Using hardware thread does not show any advantage at the beginning when data size is small (Data Size = 4K Bytes), as the large overhead on thread creation. With the data size increasing, the time for software thread processing increases rapidly, the hardware thread outperforms software thread by achieving an overall speedup of up to 20x when data size amounts to 256K. The comparison also indicates the difference on buffer allocation. Although Hardware Thread with Dynamic Buffer Allocation Support (HT with DBAS) is a little inferior to Hardware Thread with Static Buffer Allocation Support (HT with SBAS) in performance, its potential programming transparency is greater than the HT with SBAS.

V. CONCLUSIONS AND FUTURE WORK

In order to provide a unified hardware/software thread programming model for stream driven applications running on DRSoC, a standard POSIX-compliant HTI and a portable stub/interface adaptation are proposed in this paper. During the process of interface design and adaptation, the software stub thread is used on behalf of the hardware thread, making shared buffer based multi-thread programming portable and efficient. The experimental results on AES encryption/decryption show that both the resource utilization and communication overhead of hardware thread adaptation are at low level. The designers could exploit the unified multithread programming transparency while effectively maintain hardware performance.

The communication/synchronization request of hardware thread is now transferred to its stub thread according to our stub/interface adaptation mechanism, which incurs long overhead and hides the inherent speedup of stream driven application. Future research will focus on building a direct communication channel and hardware based semaphore, making inter-thread communication/synchronization between two hardware threads more efficient while preserving programming transparency.

REFERENCES


