SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model

Ying Wang, Xuegong Zhou, Lingli Wang, *Member, IEEE*,
Jian Yan, Wayne Luk, *Fellow, IEEE*, Chenglian Peng, and Jiarong Tong

Abstract—**Partially reconfigurable systems are promising computing platforms for streaming applications, which demand both hardware efficiency and reconfigurable flexibility. To realize the full potential of these systems, a streaming-based partially reconfigurable architecture and unified software/hardware multithreaded programming model (SPREAD) is presented in this paper. SPREAD is a reconfigurable architecture with a unified software/hardware thread interface and high throughput point-to-point streaming structure. It supports dynamic computing resource allocation, runtime software/hardware switching, and streaming-based multithreaded management at the operating system level. SPREAD is designed to provide programmers of streaming applications with a unified view of threads, allowing them to exploit thread, data, and pipeline parallelism; it enhances hardware efficiency while simplifying the development of streaming applications for partially reconfigurable systems. Experimental results targeting cryptography applications demonstrate the feasibility and superior performance of SPREAD. Moreover, the parallelized Advanced Encryption Standard (AES), Data Encryption Standard (DES), and Triple DES (3DES) hardware threads on field-programmable gate arrays show 1.61–4.59 times higher power efficiency than their implementations on state-of-the-art graphics processing units.**

Index Terms—**Hardware thread, parallelism, partial reconfiguration, streaming application.**

I. Introduction

DURING the last decade, embedded processor and reconfigurable processing units (RPUs) have been integrated within field-programmable gate arrays (FPGAs) to form partially reconfigurable systems. On these systems, streaming applications, which commonly appear in the context of multimedia processing, digital signal processing, and data encryption/decryption have demonstrated performance improvement and reconfigurable flexibility [1]–[4]. However, there are still challenges in designing these applications. First, with the increasing need for both higher performance and design flexibility, how do we combine partial reconfigurability with streaming structure on a chip? Second, since a computation kernel or a task can be implemented in software or hardware, how do we enable switching between software and hardware to improve runtime adaptability? Third, as the number of RPU increases, the spatial organization of streaming channels among RPUs supports different kinds of parallelism; how do we expose such parallelism to application programmers for further performance improvement?

To address these challenges, SPREAD provides a hierarchical software/hardware (SW/HW) co-design solution specifically designed for streaming applications. At the architecture level, we propose a high throughput point-to-point streaming structure, where a streaming computation can be represented as a set of software threads and hardware threads that communicate explicitly over streaming channels. A unified hardware thread interface (HTI), coupled with a novel method based on "switchable threads" and "stub threads," allows hardware threads to be managed in the same way as software threads, and enables seamlessly online switching between software and hardware implementations. At the operating system level, a lightweight operating system kernel has been extended for dynamic computing resource allocation and streaming-based multithread management. Moreover, an extended SW/HW multithreaded programming library is provided to improve design productivity and SW/HW switching adaptability. It is easy for programmers to design streaming applications and exploit the inherent thread, data, and pipeline parallelism with the aid of this library.

A case study involving data encryption/decryption applications is used to demonstrate the feasibility and SW/HW switching adaptability of SPREAD. Experimental results indicated that the resource overhead of the HTI is acceptable. Hardware threads show superior performance compared with accelerators using fast simplex link (FSL) or system bus. Moreover, the power efficiency of the AES/DES/3DES hardware thread is found to be superior to the corresponding implementations on state-of-the-art graphics processing units (GPUs) by making use of different kinds of parallelism.

Manuscript received April 19, 2012; revised October 5, 2012; accepted October 30, 2012. Date of publication January 11, 2013; date of current version October 14, 2013. This work was supported in part by the National Natural Science Foundation of China under Grant 61131001 and Grant 61171011, the National 863 Program of China under Grant 2009AA012201, and the State Key Laboratory of ASIC and System Research Program of Fudan University under Grant 09XT004 and Grant 09ZD005, the U.K. Engineering and Physical Sciences Research Council, and the European Union Seventh Framework Program under Grant 248976, Grant 257906, Grant 287804, and Grant 318521.

Y. Wang, X. Zhou, L. Wang, J. Yan, and J. Tong are with the State Key Laboratory of ASIC and System, Fudan University, Shanghai 201203, China (e-mail: ying_w@fudan.edu.cn; zhouxg@fudan.edu.cn; llwang@fudan.edu.cn; 11210720144@fudan.edu.cn; jrtong@fudan.edu.cn).

W. Luk is with the Department of Computing, Imperial College London, London SW7 2AZ, U.K. (e-mail: wl@doc.ic.ac.uk).

C. Peng is with the School of Computer Science and Technology, Fudan University, Shanghai 201203, China (e-mail: clpeng@fudan.edu.cn).

Digital Object Identifier 10.1109/TVLSI.2012.2231101

To summarize, the following contributions are made in this paper.

1) A high-throughput point-to-point streaming channel is proposed. These channels can be dynamically interconnected according to thread dependency, making partially reconfigurable architecture simple and efficient for streaming applications.
2) A method based on switchable threads and stub threads to enable smooth switching between software and hardware implementations. Such switching can be achieved through dynamic resource allocation, fast context transfer, and stream redirection to improve runtime adaptability.
3) An extended stream programming library for both software threads and hardware threads, which facilitates the design of streaming applications in a unified SW/HW thread model, and allows exploitation of thread, data, and pipeline parallelism for further performance improvement.
4) An evaluation of the proposed approach based on cryptographic designs on an XC4VFX60 FPGA, showing that SPREAD offers higher throughput and higher power efficiency than implementations on CPU and digital signal processing (DSP) chips. Compared with GPUs, SPREAD is capable of higher power efficiency of 1.61 to 4.59 times.

This paper is arranged as follows. Section II discusses related work. Section III gives overall design considerations. Section IV presents hierarchical design of SPREAD in detail. Section V provides experimental results and analysis. Section VI compares our work with other solutions. Finally, Section VII concludes this paper.

II. Related Work

Efficient architectures and portable programming models play an important role for streaming applications. For heterogeneous reconfigurable systems, including CPU and RPUs, several solutions have been proposed at different abstraction levels. Some solutions are based on specific compilers and predefined libraries/streaming languages [5]–[8], which contribute to achieving a working prototype much faster but do not support partial reconfiguration. SCORE is a programming model, which combines pipe-and-filter architecture, customized compiler, and runtime support [9]; it has not been implemented in FPGA platforms.

Several groups focus on unified SW/HW programming models with different extensions. A virtualization layer and the corresponding operating system architecture have been proposed for reconfigurable applications [10]. An extended Linux kernel has been introduced in BORPH to support a unified file system access for SW/HW tasks [11]. An hthread programming model has been developed [12], where key operating system services of inter-thread synchronization and thread scheduling are migrated to hardware units, providing equal access to both hardware and software threads. Recently, a framework named FUSE is proposed at the operating system level, where hardware tasks are viewed as memory-mapped I/O devices [13]. However, these extensions do not address several important aspects for stream-oriented partially reconfigurable systems.

1) A high throughput streaming channel support at structure level. Although BORPH supports file streaming I/O, the bandwidth of the streaming channel is limited. FUSE and hthread are based on system bus architecture, which are unsuitable for streaming computations.
2) SW/HW switching support to improve runtime adaptability, which is not addressed by these solutions.
3) The structure and programming model should support exploitation of different kinds of parallelism, which would contribute to further performance improvement of streaming applications.

An effort related to ours is ReconOS [14], an extended pthread programming model introduced as a general solution for reconfigurable applications, which provides a rich set of operating system functions for hardware threads. Unlike ReconOS, SPREAD is an integrated solution specifically developed for streaming application design and runtime management. Hardware threads designed with SPREAD can be dynamically created, terminated, or switched to software implementations, thus increasing runtime flexibility and RPU resource utilization. Compared with statically allocated FIFO module between hardware threads, the streaming channels in our architecture can be dynamically interconnected at runtime, providing efficient inter-thread communication and support for exploiting multiple kinds of parallelism.

Furthermore, although the idea of a unified SW/HW thread abstraction is employed, we focus on providing an extended streaming communication support in a unified way for both software threads and hardware threads. Our approach takes into account characteristics of a stream processing model, and supports high throughput inter-thread communication and explicit parallelism description in streaming applications.

III. Design Considerations

A streaming application often organizes data as streams and carries out computation by kernels. Partially reconfigurable systems are promising computation platforms for streaming applications, where kernels can be designed as hardware threads for high performance and reconfigurable flexibility.

From the view of stream processing, the concept of thread provides a suitable abstraction for computation kernels. First, the behavior of a hardware thread has a significant similarity with a software thread. Once created or reconfigured, it enters an active state until exit. It is easy for programmers to describe software/hardware partitioning in a unified thread view. Second, after entering the active state, a hardware thread operates on a long sequence of data, i.e., a data stream. There are three regular execution phases: data reading, data processing, and result writing. These phases are the same as software threads, offering an opportunity for runtime SW/HW switching. Third, streaming applications can be represented as a composition of software and hardware threads, which can be managed in a unified way at the operating system level. One thread can communicate with another thread explicitly over

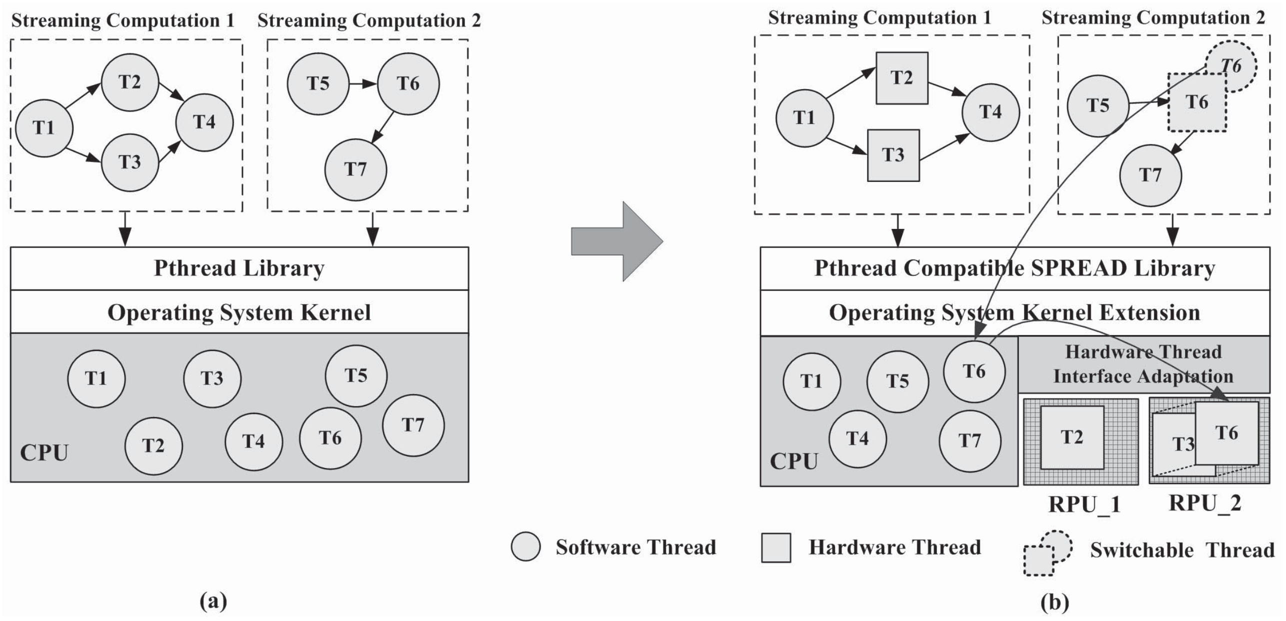


Fig. 1. Comparison between (a) pthread programming model and (b) SPREAD model.

data-driven streaming channels, and either or both threads can be implemented in software or in hardware to simplify inter-thread communication and synchronization.

Based on the above considerations, computation kernels can be abstracted to concurrently running hardware threads. Compared with the traditional pthread programming model [15] illustrated in Fig. 1(a), the SPREAD model shown in Fig. 1(b) is built on a partially reconfigurable architecture where software threads run on the CPU, while hardware threads run on the RPUs. Through streaming interface adaptation, hardware threads can be managed by an extended operating system kernel, and a high throughput streaming channel can be used for inter-thread communication. The extended operating system kernel is provided for stream management and dynamic resource allocation to improve runtime adaptability and RPU resource utilization. Furthermore, SPREAD also includes pthread-compatible APIs and an extended stream programming library for both software threads and hardware threads.

To illustrate the SPREAD approach, consider the two streaming computations shown in Fig. 1(b). One computation consists of four threads organized as a split-join structure, while the other consists of three threads organized as a pipeline structure. The SPREAD library mentioned above enables programmers to specify the $T2$, $T3$, and $T6$ tasks in these two computations as hardware threads. All running threads, whether hardware or software ones, are under the control of the extended operating kernel.

During the development of applications, we adopt switchable threads, which are a special kind of hardware thread, to facilitate switching between software and hardware at runtime. A switchable thread includes a pair of hardware and software implementations with the same behavior, so that its implementation style can be dynamically changed depending on application or environment requirements. The SW/HW switching can be achieved via dynamic resource allocation, fast context transfer, and stream redirection. Taking the scenario in Fig. 1(b) as an example, $T6$ is defined as a switchable thread according to initial software/hardware partitioning at design time. When it is created, if no RPU resources are available, and all running hardware threads are unswitchable, then the software implementation of $T6$ is chosen. Once resource RPU_2 is released by hardware thread $T3$, the software implementation of $T6$ can be switched to its hardware implementation with the aid of SPREAD.

To enable the extended operating system kernel to manage hardware threads at runtime, we propose an approach based on the idea of a stub thread. As the software delegate of a hardware thread, a stub thread is created at the same time as its corresponding hardware thread. A stub thread can be used to redirect the streaming communication primitives within the extended operating system kernel to a specific hardware thread, and to monitor HTI. For a switchable thread, the software and hardware implementations with identical functional behavior coexist within a stub thread, which makes SW/HW switching easy and controllable. The streaming-based HTI, coupled with a stub thread, allows hardware threads to be managed via an extended operating system kernel in the same way as software threads, while keeping streaming threads running efficiently. The design of a stub thread will be explained in Section IV-D.

As the dynamically interconnected streaming channels represent different streaming structures, it is easy for programmers to leverage thread parallelism, data parallelism, and pipeline parallelism in a unified view of threads. Thread parallelism can be naturally exploited for streaming applications, which consist of software threads and hardware threads running independently and concurrently. Data parallelism can be exploited by allocating RPUs to duplicated hardware threads and programming them in a single program multiple data (SPMD) fashion. SPREAD also allows programmers to create a producer thread and a consumer thread in a pipeline parallel fashion. The ability to exploit multiple kinds of parallelism

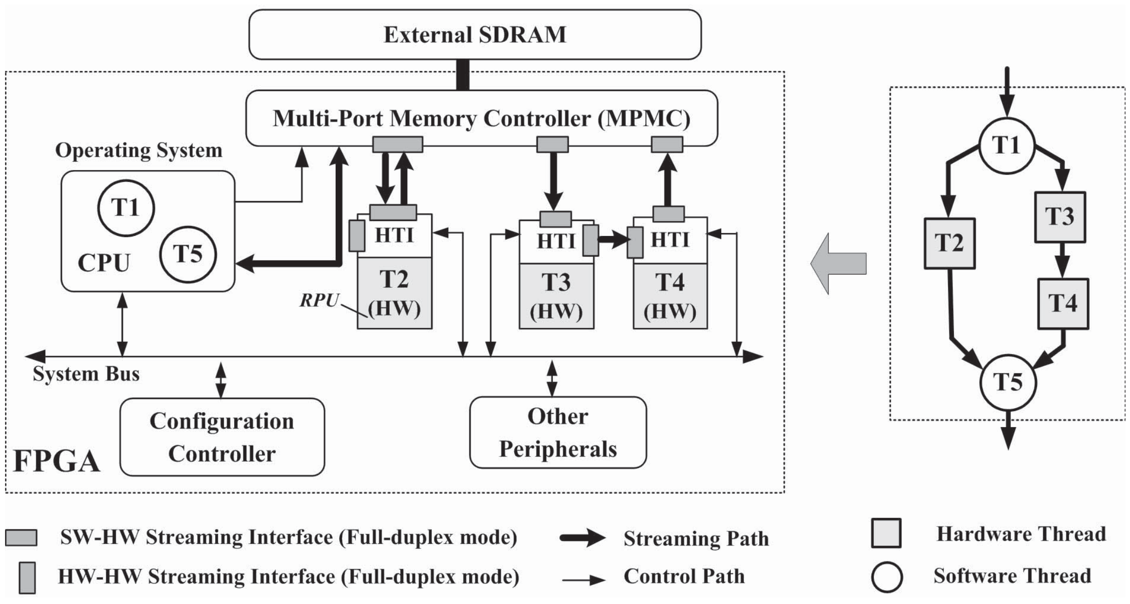


Fig. 2. Streaming-based partially reconfigurable system architecture.

enhances programming flexibility and performance through improved utilization of RPU resources and point-to-point streaming interconnections.

IV. Hierarchical Design of SPREAD

Fig. 1(b) provides an overview of SPREAD for streaming applications. This section will introduce each part of SPREAD in detail.

A. Partially Reconfigurable System Architecture

Fig. 2 depicts the architecture of a streaming-based partially reconfigurable system, which consists of a CPU, several RPUs, an external SDRAM, a configuration controller, and other peripherals. Software threads and the operating system are running on the CPU. Compared with the CPU, RPUs are reconfigurable computing resources for coarse-grained hardware threads, which can be dynamically created, terminated, or switched during runtime.

In order to support high throughput communication for stream-oriented hardware threads, a control interface and two streaming interfaces are provided separately in a hardware thread interface. The bus-based control interface is used for SW/HW switching and status monitoring. Two streaming interfaces are provided for full-duplex, synchronous point-to-point communication, through which a hardware thread can communicate with another thread, whether software or hardware, in an efficient way.

For SPREAD, a FIFO is employed as the local memory of each hardware thread, while the external SDRAM region is allocated for global data sources and storage of computation results. The inter-thread communication can be implemented based on a predefined streaming channel through the point-to-point streaming interconnection. In addition, this point-to-point streaming interconnection can be dynamically configured at runtime according to data dependency among threads. For example, the interconnection configured on the left side of

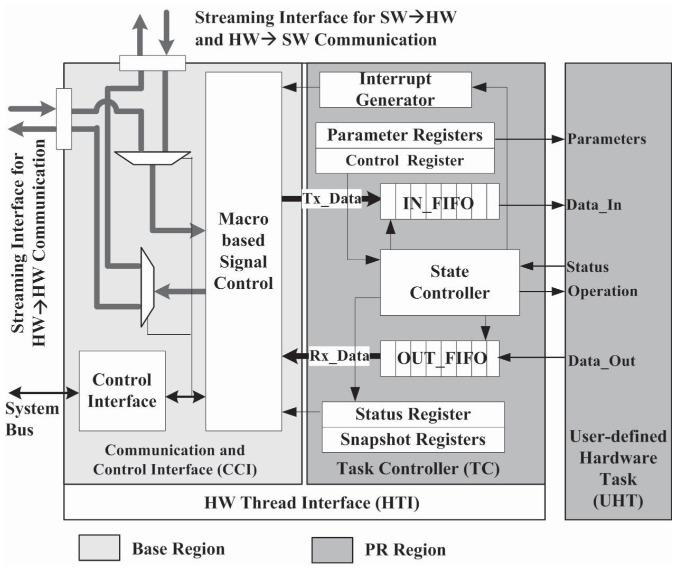


Fig. 3. HTI architecture.

Fig. 2 can be used to implement the streaming computation on the right side. Apart from the simultaneous access issue, a streaming path eliminates the dataflow bottleneck commonly found in a system bus, making our partially reconfigurable architecture simple and efficient for streaming applications.

The architecture proposed here makes it easy to configure streaming threads in different programming structures, such as split-join, pipeline, and feedback loop. Furthermore, the proposed HTI not only supports hardware thread customization and high throughput communication, but also facilitates thread, data, and pipeline parallelism exploitation.

B. Hardware Thread Customization and Adaptation

A partially reconfigurable system typically comprises an area (base region) for a static base system and one or more

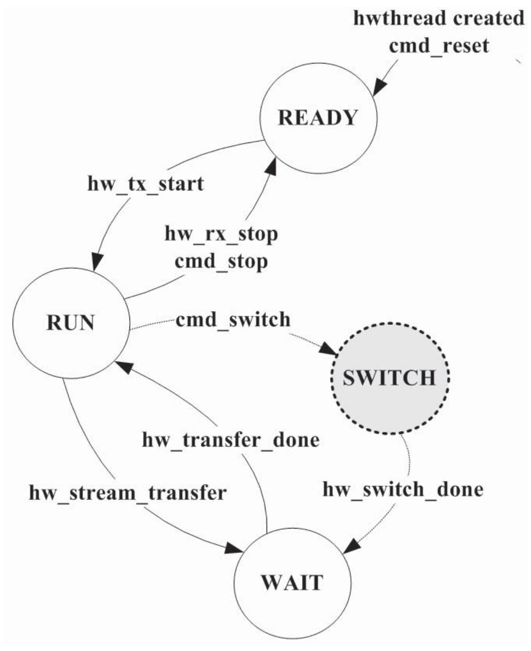


Fig. 4. State transitions of a running hardware thread.

partially reconfigurable regions (PR regions) for RPUs, such that hardware threads are executed on RPUs. For a given user-defined hardware task (UHT), a unified HTI is proposed to integrate a UHT into the system without incurring a significant performance loss. This stream-oriented HTI structure supports efficient inter-thread communication, while facilitating the unified thread management and runtime SW/HW switching.

The HTI consists of the communication and control interface (CCI) in the base region and the task controller (TC) in the PR region, as shown in Fig. 3. The communication macros within CCI provide static routing channels. A hardware thread can connect to these routing channels via predefined connection points for communication. Besides a bus interface for thread control and status monitoring, there are two point-to-point streaming interfaces covering both communication between two hardware threads, and communication between a hardware thread and a software thread. The data streams that a hardware thread operates on are grouped into packets, which consist of a large sequence of data items. A data item is the smallest unit of computation in a UHT. The TC is responsible for dealing with stream computation behavior and SW/HW switching at runtime.

At the heart of the HTI, the State Controller realizes the possible state transitions according to the pthread-compatible semantics, READY, RUN, and WAIT. A running hardware thread reads data from IN_FIFO, performs computation, and writes results to OUT_FIFO. IN_FIFO and OUT_FIFO are provided for data buffering, while enabling blocking to take place in inter-thread communication.

The state transfer of a running hardware thread is described in Fig. 4. Since a data-driven hardware thread is characterized by regular processing of data streams, there are mainly four states in the State Controller, which are similar to that of a software thread. A hardware thread enters into the READY state after successfully created or reset, and it will turn into the RUN state upon receiving a start of packet transfer signal (hw_tx_start). While in the RUN state, when IN_FIFO is empty or OUT_FIFO is full, a hardware thread will immediately assert an hw_stream_transfer signal, blocking itself, and entering into the WAIT state. A hardware thread will return to the RUN state upon receiving an hw_transfer_done signal. Additionally, a hardware thread could go back to the READY state upon receiving a cmd_stop command from the operating system kernel or an hw_rx_stop signal from the streaming interface.

For switchable threads that could change their implementation style from hardware to software, the State Controller also provides SW/HW switching support. Since each data item expires after being processed, the hardware thread context can be captured using the OUT_FIFO and the snapshot registers. The snapshot registers include a data item counter to show runtime progress and thread parameters. While in the SWITCH state, snapshot registers can be read out together with the processed data items within the OUT_FIFO.

The hardware thread customization flow is provided in SPREAD. First, a given UHT in the form of an RTL code or a netlist is integrated with the TC to produce a hardware thread entity. Design optimizations for on-chip communication infrastructure can also be included [16]. Second, the proposed streaming structure with point-to-point interconnections and the hardware thread entities are synthesized with RTL synthesis tools. The output netlists of the base system and the hardware threads, as well as RPU layout information, constitute the input to a partially reconfigurable system design tool, such as Xilinx PlanAhead. Third, the full bitstream of the system and partial bitstreams of the hardware threads are produced. The partial bitstreams can be dynamically loaded when creating a hardware thread at runtime.

C. Operating System Kernel Extension

Since traditional operating systems regard hardware tasks as devices, streaming application programmers are required not only to manipulate inter-task communication and synchronization according to the customized device driver, but also to deal with issues of resource allocation and task management. Thus, it is difficult to exploit different kinds of parallelism, partial reconfigurable flexibility, and SW/HW switching adaptability.

Hardware threads in streaming applications form a virtual hardware thread library where partial bitstreams targeting all available RPUs are stored. The dynamic resource allocation can be achieved by means of one-to-many mappings between a hardware thread and the available RPUs. Based on these considerations, an extended operating system kernel is provided to manage computing resources, hardware threads, and streams.

1) Reconfigurable Computing Resource Management: In our approach, the RPUs shown in Fig. 2 are regarded as reconfigurable computing resources, which can be allocated to hardware threads (including switchable threads) during runtime.

An extended resource manager is responsible for tracking the changing status of the RPUs. An Idle RPU can be

Algorithm 1 Definition of the HTCB

```
typedef struct spread_htcb_s {
    bool           m_bSwitchable;  // 0- unswitchable, 1-switchable
    unsigned int  m_nImpType;
    // switchable thread implementation style: 0-SW, 1-HW
    unsigned int  m_nHWThreadID;
    unsigned int  m_nHWThreadState;
    // 0-Idle, 1-New, 2-Switching, 3-Active
    char*          m_pHWThreadAddr;  //bitstream file address
    unsigned int  m_nRPUID;  // allocated RPU ID
} spread_htcb_t;
```

dynamically allocated to a newly created hardware thread without manual intervention. Once a hardware thread is successfully created, the corresponding RPU will be transferred from the Configured state to the Allocated state. This RPU will be released and returned to the Idle state after the hardware thread finishes. Configuration hit would occur when the released RPU is reallocated to the same hardware thread executed before.

A partially reconfigurable architecture would have a large impact on the throughput of streaming computations, especially when hardware threads are created at runtime. A configuration cache mechanism is employed to reduce hardware thread creation time. Considering possible configuration hits during runtime, the identification of a hardware thread is recorded in the resource allocation table (RAT). If a configuration hit takes place when creating a hardware thread, there is no need to load the partial bitstream and the hardware thread creation time will be significantly decreased. Otherwise, a configuration miss occurs and the RPU will be reconfigured with the corresponding partial bitstream.

2) Hardware Thread Management: A streaming application consists of software threads and hardware threads. It is necessary to manage all software and hardware threads based upon our partially reconfigurable architecture.

The hardware thread manager is built on the extended hardware thread control block (HTCB), which tracks state changes of all hardware threads. The HTCB data structure is shown in Algorithm 1. The pointer to HTCB is added to the pthread structure in the operating system kernel, which simplifies hardware thread management.

3) Stream Management: The data stream is built on a full duplex streaming channel between two threads. In order to expose streams and their logic connections to application programmers, stream management is offered as a service for software threads and hardware threads.

The stream manager allows flexible point-to-point interconnection between two threads, and provides support for streaming communication. It supports the FIFO style buffer and the corresponding semaphores for streaming communication between two software threads. At the same time, the LocalLink interface [17] and buffer descriptor manipulation are employed for SW$\rightarrow$HW, HW$\rightarrow$SW, and HW$\rightarrow$HW streaming communications.

D. Pthread-Compatible SPREAD Programming Model

Based on the extended operating system kernel, a unified SW/HW multithreaded programming model is proposed for hardware thread creation and termination, streaming communication as well as SW/HW switching. Through a light-weight programming library, programmers can describe pipeline, split-join, and feedback loop stream structures easily through dynamically established streaming channels, and exploit the inherent parallelism and runtime SW/HW switching in streaming applications.

The essential programming support of SPREAD is described below.

1) Hardware Thread Creation and Termination: It is the responsibility of the extended hardware thread manager to deal with the hardware thread creation and termination. Dynamic resource allocation occurs when a hardware thread is created via API hwthread_create(). If configuration hits, the released RPU is allocated immediately with configuration reuse. If configuration misses, an RPU is allocated to a newly created hardware thread according to its attribute settings. It is assumed that the priority of an unswitchable hardware thread is higher than that of a switchable thread.

If a RPU is successfully allocated to a hardware thread, the partial bitstream of this hardware thread will be loaded and the corresponding stub thread will be created. If there is no RPU resource available for unswitchable thread creation, API hwthread_create() will search for the hardware implementation of the switchable thread for resource preemption. The implementation style of the preempted switchable thread can be changed through its stub thread. A successfully created hardware thread will stay in active hardware thread list until termination. It will be added to the idle hardware thread list on termination for the sake of configuration reuse.

For a newly created switchable thread, if there is no RPU resource available, it will turn to software implementation. When an RPU resource is available, this software implemented switchable thread is scheduled for hardware execution according to the priority scheduling policy.

2) Streaming Communication Support: Streams provide a simple and efficient way of communication for software threads and hardware threads. As depicted in Algorithm 2, API stream_create() creates a streaming channel for a pair of threads, and returns an identifier. Thus, two threads, whether implemented in hardware or in software, can operate on a streaming channel and are coordinated based on the same stream identifier. The following are possible results when a data packet is transferred from the sender thread to the receiver thread.

1) If both threads are implemented in software, the sender thread first puts the data packet into the predefined data FIFO via the available memcpy() function, then the receiver thread gets the data packet from this data FIFO by memcpy(). The inter-thread synchronization is based on traditional semaphore support with FIFO "empty" and "full" status.
2) If two threads are implemented in hardware and software, a synchronous, point-to-point connection, and a

Algorithm 2 Stream Creation Process

```
int stream_create(pthread_t &ti, pthread_t &tj, int PortNum)
{
    spread_stream  *stream;
    stream = (spread_stream *)malloc (sizeof (spread_stream));
     if(ti and tj are implemented in hardware)
        stream_chan_hw (ti,tj,PortNum); // HW→HW streaming
     else if(ti and tj are implemented in software)
        stream_chan_sw_init(ti,tj,PortNum);
         // SW→SW streaming
    else
         stream_chan_dma(ti,tj,PortNum);
         // SW→HW or HW→SW streaming

    if(stream is built successfully)
          return stream-> stream_id;
    else
         return -1;
}
```

DMA engine can be used for transmitting and receiving data packets on a stream. The synchronization here depends on the built-in synchronous streaming interface in HTI.

3) When a data packet is directly transferred from one hardware thread to another, as the streaming interfaces are synchronized by virtue of the FIFO within each HTI, the streaming channel is dynamically configured when a stream is created.

After a stream is successfully created, the programmer can aggregate data items into packets before the transmission starts. To support efficient communication between threads, SPREAD offers a push/pull mechanism for moving high-volume data streams. Fig. 5 shows a SPREAD-based multi threaded example, involving AES encryption, where one software thread produces the data while the unswitchable AES hardware thread encrypts the data; another software thread consumes the data.

Within the main process shown in Fig. 5, the streaming channel SC1 and SC2 are dynamically created. Programmers can use API stream_open() to request streaming communication service and open a particular streaming channel in either read or write mode. Once the stream is opened and acknowledged, the producer thread uses API stream_out() to write a data packet to a previously opened stream, while the consumer thread uses API stream_in() to read a data packet from the stream. The size of the data packet to be sent is also passed as an argument. For unswitchable hardware thread responsible for AES data encryption, the stream input and output operations are automatically implemented within HTI, thus the stub thread only needs to perform status monitoring.

3) Stub Thread Enabled SW/HW Switching: When designing streaming applications, there are two possibilities. One is that a computation kernel can be implemented in software and hardware, and the implementation style of this kernel is not determined at design time on account of the changing environments. The other is that the kernel depends on other software threads for data input and output. Under these conditions, the computation kernel can be created as a switchable thread. At run time, a switchable hardware thread with low priority can be preempted by another hardware thread with higher priority to meet the real-time constraints.

In SPREAD, a stub thread enabled SW/HW switching method is used to improve runtime adaptability. The stub thread itself provides a unified wrapper for the hardware and software implementations within a switchable thread. There are three states in a stub thread: 1) a switchable thread running in hardware; 2) SW/HW switching; and 3) a switchable thread running in software. SW/HW switching occurs if resource preemption occurs during the process of RPU allocation or the required RPU resource is released. Programmers need to deal with thread-specific context switching and stream redirection, and add the necessary synchronization between the hardware thread manager and the stub thread. However, frequent changes in implementation style (i.e., switch thrashing) will result in significant decrease in performance and increase in CPU utilization. In order to prevent switch thrashing, the number of switches is restricted to one in the hardware thread manager.

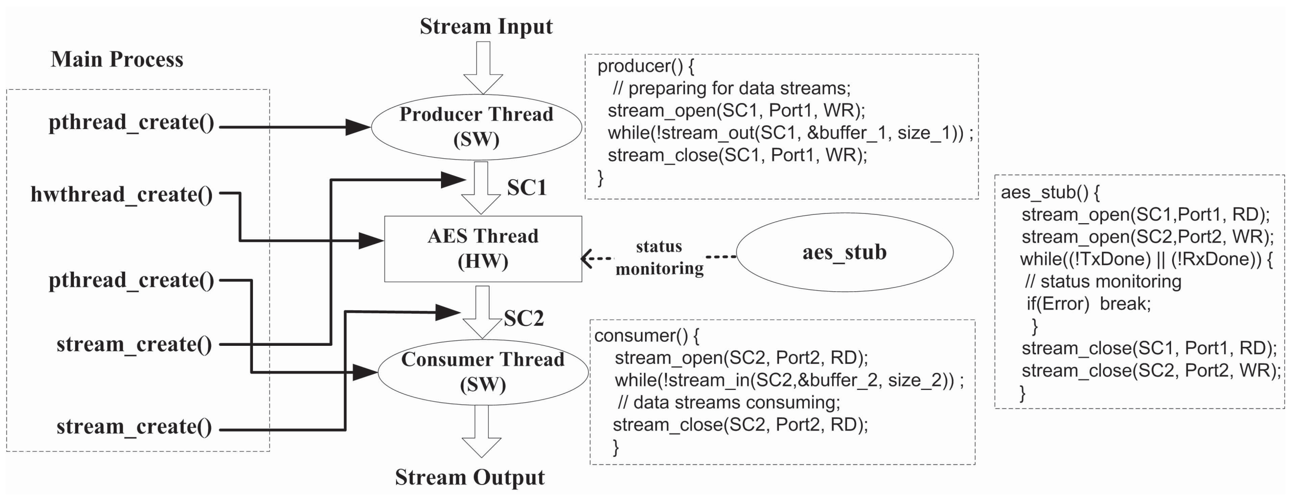
An example of switchable hardware thread on AES encryption is shown in Fig. 6, where stub thread enabled HW→SW switching is described. After receiving a switch notification H2S_Switch from the hardware thread manager, the stub thread immediately sends a cmd_switch signal to the switchable thread running in hardware, and reads the snapshot registers shown in Fig. 3. The switchable thread in hardware then enters the "SWITCH" state and initiates context switching. The context switching method is based on the sliding-window pattern of streaming computation. Thus, there is no need to deal with low-level state changes within the UHTs [18]. The stream-oriented thread context consists of data items that have already been processed and data stored in user-defined snapshot registers, which are transferred through the streaming channel. When the context transfer is completed, the stub thread will update the pointer to the stream buffer, and change the stream address reference from SC1, SC2 to newly created SC3 and SC4 (i.e., stream redirection). The use of stub threads for stream update and redirection enables the implementation style of a switchable thread to be seamlessly changed from hardware to software and vice versa.

SW→HW switching is similar to the process discussed above. The main difference is that the thread manager should configure the hardware thread before sending the SW→HW switching notification.

V. Experimental Results and Analysis

To evaluate the flexibility and performance of streaming applications designed with SPREAD, a partially reconfigurable system prototype has been developed on our custom prototype board. Stream-driven cryptography applications are implemented as test cases. Fig. 7 shows the architecture of our system prototype and the FPGA floorplan.

The prototype is implemented in the Xilinx XC4VFX60 FPGA, where a PowerPC405 CPU, a multiport memory controller (MPMC), several peripherals on the processor local

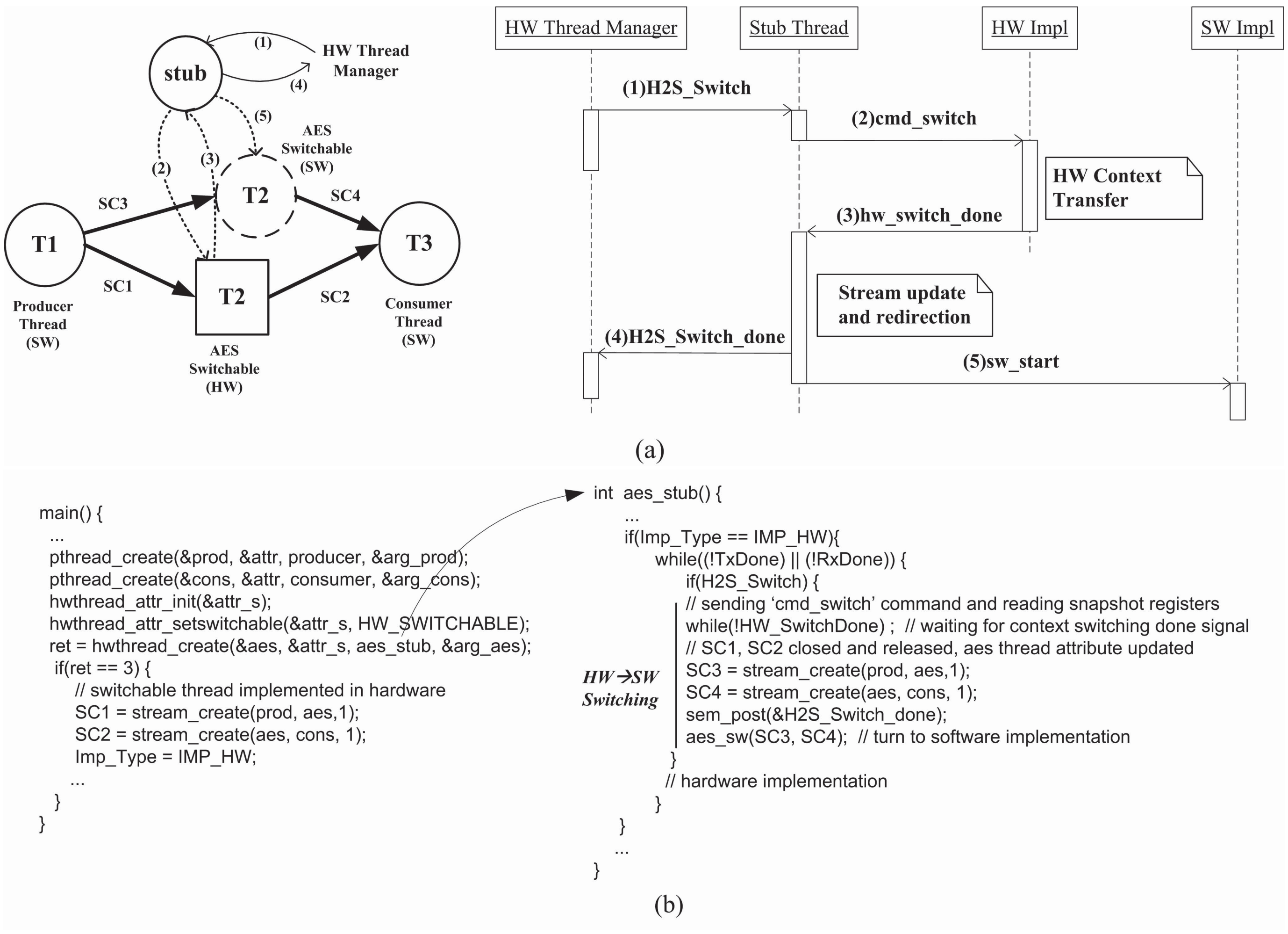


```
producer() {
   // preparing for data streams;
   stream_open(SC1, Port1, WR);
   while(!stream_out(SC1, &buffer_1, size_1)) ;
   stream_close(SC1, Port1, WR);
}
```

```
aes_stub() {
    stream_open(SC1,Port1, RD);
    stream_open(SC2,Port2, WR);
    while((!TxDone) || (!RxDone)) {
     // status monitoring
     if(Error)  break;
     }
    stream_close(SC1, Port1, RD);
    stream_close(SC2, Port2, WR);
   }
```

```
consumer() {
   stream_open(SC2, Port2, RD);
   while(!stream_in(SC2,&buffer_2, size_2)) ;
    // data streams consuming;
   stream_close(SC2, Port2, RD);
  }
```

Fig. 5. Multithreaded example on AES encryption with SPREAD.



```
main() {
  ...
  pthread_create(&prod, &attr, producer, &arg_prod);
  pthread_create(&cons, &attr, consumer, &arg_cons);
  hwthread_attr_init(&attr_s);
  hwthread_attr_setswitchable(&attr_s, HW_SWITCHABLE);
  ret = hwthread_create(&aes, &attr_s, aes_stub, &arg_aes);
   if(ret == 3) {
      // switchable thread implemented in hardware
      SC1 = stream_create(prod, aes,1);
      SC2 = stream_create(aes, cons, 1);
      Imp_Type = IMP_HW;
      ...
   }
}
```

```
int  aes_stub() {
     ...
     if(Imp_Type == IMP_HW){
          while((!TxDone) || (!RxDone)) {
               if(H2S_Switch) {
               // sending 'cmd_switch' command and reading snapshot registers
               while(!HW_SwitchDone) ;  // waiting for context switching done signal
               // SC1, SC2 closed and released, aes thread attribute updated
               SC3 = stream_create(prod, aes,1);
               SC4 = stream_create(aes, cons, 1);
               sem_post(&H2S_Switch_done);
               aes_sw(SC3, SC4);  // turn to software implementation
               }
          // hardware implementation
          }
     }
     ...
}
```

Fig. 6. Example of switchable hardware thread on AES encryption. (a) Stub thread enabled HW→SW switching. (b) Pseudo code of main process and the stub thread.

bus (PLB), as well as hardware threads are all clocked at 100 MHz. Using the Xilinx EAPR design flow [19], the prototype is designed with three RPUs and a base system. The communication between an RPU and a base system is through predefined bus macros. The UHT is abstracted to the hardware thread through the HTI. The point-to-point streaming interface within the HTI makes a hardware thread tightly coupled to its corresponding sender or receiver thread, bypassing the traditional bottlenecks of the on-chip system bus. The partial bitstreams of AES (aes_encryption), AES_INV (aes_decryption),

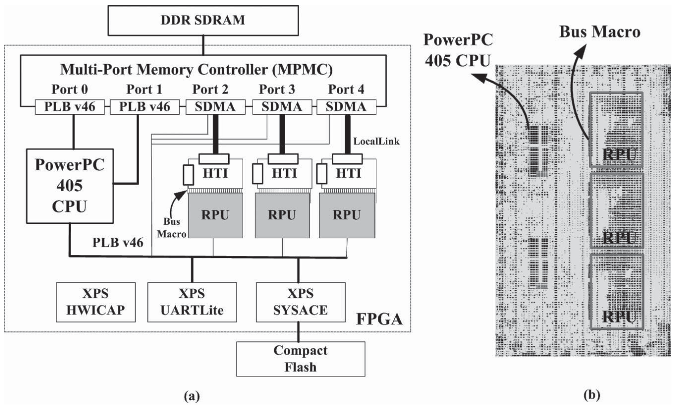


Fig. 7. Partially reconfigurable system prototype. (a) System architecture. (b) FPGA floorplan.

TABLE I
RESOURCE UTILIZATION OF A HTI

Name	Number of Slices	Number of FF	Number of LUT	Number of BRAM
HTI	488	723	706	2
XC4VFX60	25280	50560	50560	232
Utilization	1.93%	1.43%	1.4%	0.86%

and DES (des_encryption/des_decryption) hardware threads are produced through the thread customization flow. These partial bitstreams are stored in the Compact Flash, and then transferred to the SDRAM at system start-up. The HWICAP is used for hardware thread configuration at runtime.

The Xilkernel 4.0 [20] is extended to provide RPU resource management, multithreaded, and stream management. Hardware threads are capable of time-sharing the available RPUs. A hardware thread can be dynamically created, terminated, or switched with the support of SPREAD. The SW/HW switching method can adapt to the changing environment during runtime. Moreover, the proposed stream-oriented HTI and synchronous point-to-point connection contribute to different kinds of parallelism exploitation for further performance improvement.

A. HTI Resources Utilization

As shown in Table I, the resources used for an HTI are all kept at a low level, accounting for 1.93%, 1.43%, 1.4%, and 0.86% of the total numbers of slices, flip flops, LUTs, and BRAMs on the XC4VFX60, respectively.

B. Hardware Thread Creation Time

The creation time of a hardware thread consists of the overhead on thread configuration, thread management as well as the corresponding stub thread creation.

Table II gives the total creation time of three hardware threads within our partially reconfigurable system prototype. It can be seen that a large portion of the creation time is spent on loading a partial bitstream when a configuration miss occurs. The configuration time is large due to the limited throughput of HWICAP, which processes configuration bitstreams. Based on the PowerPC405 CPU with I-Cache/D-Cache enabled, the

TABLE II
HARDWARE THREADS CREATION TIME. 252758/241725/229878 BYTES IS THE SIZE OF AES/AES_INV/DES HARDWARE THREAD

	AES	AES_INV	DES
Configuration Time (ms)	22.14	21.85	20.75
HW Thread Management and Stub Thread Creation Time (ms)	0.05	0.05	0.05
HW Thread Total Creation Time with Configuration Miss (ms)	22.19	21.90	20.80
HW Thread Total Creation Time with Configuration Hit (ms)	0.05	0.05	0.05

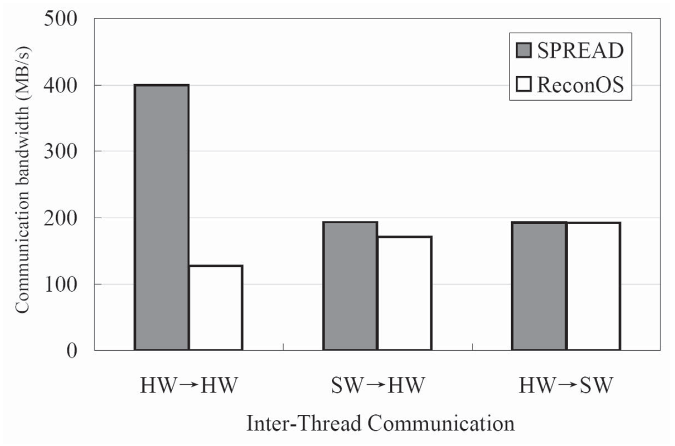


Fig. 8. Inter-thread communication bandwidth comparison between SPREAD and ReconOS.

TABLE III
TOTAL PROCESSING TIME WHEN DATA SIZE = 4M BYTES

	SW Thread with I-Cache/D-Cache Enabled (ms)	HW Thread with SPREAD (ms)	Speedup
AES	15687.68	58.65	267.48
AES_INV	24186.88	58.65	412.39
DES	4648.96	112.39	41.36

average time for a software thread creation is 19.2 us with the default settings of API pthread_create(). When a configuration hit occurs, the total creation time of a hardware thread is almost 50 us; the same order of magnitude as that of a software thread.

C. Inter-Thread Communication Bandwidth

A hardware thread runs at 100 MHz, and the IN_FIFO and OUT_FIFO within the hardware thread are organized with 32-b wide and 32-word depth. The streaming channel width is 32-b. The communication bandwidth between two hardware threads achieves 400M Bytes/s. The communication between a software thread and a hardware thread involves the SDMA controller within the MPMC, which employs the burst data transfer mechanism and reaches almost 200M Bytes/s.

According to the testing results given in [14], Fig. 8 depicts the inter-thread communication bandwidth comparison

TABLE IV

THROUGHPUT COMPARISON OF HW TASKS WITH DIFFERENT ARCHITECTURES AND TASK INTERFACES

	System Proposed by [23]	System Proposed by [24]	System Proposed by [25]		Our Prototype
	FSL based Interface	Bus based Interface	FSL based Interface	Bus based Interface	HW Thread designed with SPREAD
AES Throughput (Mbps)	70.95	261.6	31.68	10.39	545.61
DES Throughput (Mbps)	/	280.8	23.71	5.19	284.72

between SPREAD and ReconOS. The HW→HW communication bandwidth provided by SPREAD is 3.14 times faster than ReconOS with the aid of high-speed streaming interface of the hardware thread. Furthermore, the communication channel between two hardware threads is fixed in ReconOS, while the streaming channels in SPREAD can be dynamically configured for streaming applications.

Although the bandwidths of MEM→HW (burst read) and HW→MEM (burst write) in ReconOS are comparable to SW→HW and HW→SW in SPREAD, there are three main differences between these two methods. First, since both burst read and write operations in ReconOS are built on the bus master access, the available CPU bandwidth will decrease as the number of data transfer increases. SPREAD provides bus-independent, full duplex point-to-point communication among threads, which permits more concurrent data transfer while the CPU is still highly available. Second, if there are multiple hardware threads running concurrently which demand high throughput inter-thread communication, ReconOS has two limitations: serial arbitration for memory access and data transfer on the system bus. In contrast, the MPMC employed in SPREAD can simultaneously arbitrate all data transfer requirements with *a prior* knowledge to efficiently use the DDR memory. Third, in order to transfer large quantities of data to and from a hardware thread, ReconOS needs explicit synchronization on frequent data partitioning and transfers. SPREAD-based communication via a streaming channel can be implicitly synchronized by the HTI interface.

D. Hardware Thread Execution

To demonstrate the feasibility and performance of hardware threads designed with SPREAD, UHTs are provided for cryptographic hardware thread customization and adaptation. The AES task [21] takes 128-b key and 128-b data as input, performing a complete ciphering sequence in 12 cycles. The DES task [22] takes 56-b key, 64-b data as input and generates a 64-b result. It needs 16 cycles to complete a full encryption/decryption sequence. The maximum throughput of the AES/DES task is 1.02 Gb/s/381.47 Mb/s at a frequency of 100 MHz.

Once a hardware thread is dynamically created with SPREAD, there are mainly three phases for stream processing: streams reading from a sender thread, data processing, and result writing to a receiver thread. Stream processing is repeated to deal with a large amount of data according to the application requirement.

As shown in Table III, when compared with software threads using I-Cache and D-Cache, the performance of hardware threads greatly benefits from the abundant fine-grained parallelism and efficient bit-level operations offered by the FPGA, as well as the streaming support from SPREAD.

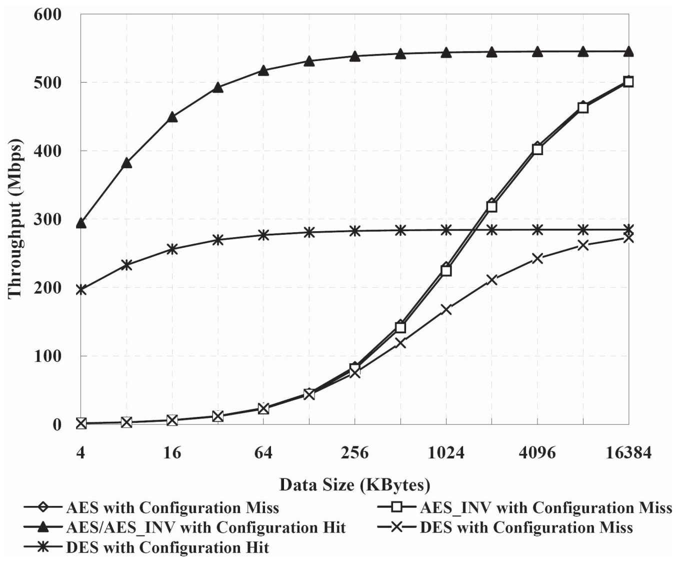


Fig. 9. Throughput of dynamically created hardware threads with the data size increased from 4 KB to 16 MB.

Table IV gives the throughput comparison of the AES and DES hardware tasks with different system architectures and task interfaces. The throughput of the AES/DES hardware thread designed with SPREAD achieves 545.61 Mb/s/284.72 Mb/s, which is 54%/75% of the maximum throughput. Our approach has demonstrated superior hardware performance over FSL-based and system bus-based hardware accelerators [23]–[25]. SPREAD can help user-defined hardware tasks achieve high performance through the stream-oriented communication interface.

As far as the whole lifecycle of hardware threads is concerned, besides the total processing time shown in Table III, the thread creation time in Table II should be added when ciphering algorithm is changed at runtime. Fig. 9 shows the throughput of a dynamically created hardware thread with the data size increased from 4 KB to 16 MB.

It can be seen that the throughput is not greatly affected by thread creation when a configuration hit occurs. The hardware thread configuration time with a configuration miss is in the range of tens of millisecond, which has a large impact on throughput especially when data size is small. The throughput of the hardware thread grows linearly as data size increases when a configuration miss arises. It is noticed that although the hardware thread with higher throughput is more sensitive

TABLE V
NUMBER OF CLOCK CYCLES NEEDED FOR SWITCHING SW/HW ON AES ENCRYPTION

	SW→HW Switching	HW→SW Switching
Context Transfer (cycles)	6834	575
Stream Update and Redirection (cycles)	1053	1280
Total (cycles)	7887	1855

to the thread creation overhead, the impact of this overhead will be reduced with the increase of data size. Additionally, as shown in Table II, there is only a slight difference between the creation time of the AES and AES_INV hardware threads, thus the throughput of these two threads shown in Fig. 9 is nearly equal when a configuration miss occurs.

E. Switchable Thread Execution

In order to illustrate the feasibility of the proposed SW/HW switching method, we have implemented two switchable threads on AES encryption and decryption. With the unified stub thread wrapper, a software implementation and a hardware implementation with identical functions are prepared for the SW/HW thread switching during runtime. Taking switchable thread on AES encryption as an example, Table V lists the number of clock cycles needed for SW/HW switching.

The total number of cycles spent on SW/HW switching is obtained by measuring the time for context transfer, stream update, and redirection. The overhead of the SW→HW switching and the HW→SW switching are 78.87 and 18.55 μs, which are acceptable for streaming applications to improve runtime adaptability. Because the software implementation usually needs more time to reach the SWITCH state when compared with the hardware implementation, the time for context transfer on the SW→HW switching is larger than the HW→SW switching.

For a switchable thread implemented in software style when created, the execution time of this switchable thread would be largely decreased if SW→HW switching occurs and a large portion of data streams is redirected to hardware processing. The proposed switching method combines the extended hardware thread manager and the stub thread, making use of the available RPU resources to improve runtime adaptability while providing performance improvement through SW→HW switching. For a newly arrived switchable thread started in hardware implementation, it can be preempted by another unswitchable thread and turns to software implementation to improve overall system responsiveness.

F. Coarse Level Parallelism Exploitation

SPREAD provides support for thread parallelism when software threads and hardware threads running independently and concurrently. Besides thread parallelism, it is easy for programmers to exploit coarse-grained data parallelism and pipeline parallelism on available RPUs. This section will present the results on the 3DES and AES implementations by exploiting pipeline parallelism and data parallelism on our prototype.

TABLE VI
3DES EXECUTION TIME COMPARISON WHEN DATE SIZE = 4M BYTES

	Implementation	Time (ms)
System Proposed by [25]	FSL based 3DES Accelerator	2443.18
	Bus based 3DES Accelerator	7444.89
FUSE [13]	3DES HW Task with LKM Loading	703.89
	3DES HW Task without LKM Loading	691.89
Our Prototype	3DES SW Thread with I-Cache/D-Cache Enabled	14387.20
	3DES Implementation with three Pipelined DES HW Threads (Configuration Miss)	179.96
	3DES Implementation with three Pipelined DES HW Threads (Configuration Hit)	112.54

Since there are insufficient resources for the 3DES design within a single RPU, we implement 3DES with three pipelined DES threads on three RPUs. Table VI lists the execution time of the different 3DES implementations for processing 4 MB of data.

More than 80 times speedup is observed for the 3DES implementation over the software thread on our prototype. The speedup is due to the support of coarse-grained pipeline parallelism with SPREAD, as well as the fine-grained parallelism within FPGA. Taking advantage of different kinds of parallelism and high throughput streaming channel, the proposed 3DES implementation provides more than an order of magnitude improvement over the FSL-based and system bus-based accelerators [25]. It also shows superior performance compared with FUSE-based 3DES hardware task [13].

AES encryption is also implemented to illustrate the support for data parallelism. Data load is uniformly distributed over concurrent running hardware threads, which are designed in an SPMD fashion. Fig. 10 gives the execution time of the parallelized AES encryption. When a configuration hit occurs, the parallelized implementation achieves significant improvement in performance, particularly for large data size. In the case of a configuration miss, the execution time of parallelized implementation with multiple threads is larger than a single thread when the data size is small. With the increase of data size, the influence of the configuration time will decrease, and the parallelized AES encryption will show performance advantage.

As far as the degree of parallelism is concerned, we observe that the speedup is nonlinear with the increasing degree of parallelism. Although all memory banks can be open at the same time, and the data source and destination are located in different memory banks, the time for data transfer is largely influenced by the increasing number of simultaneous

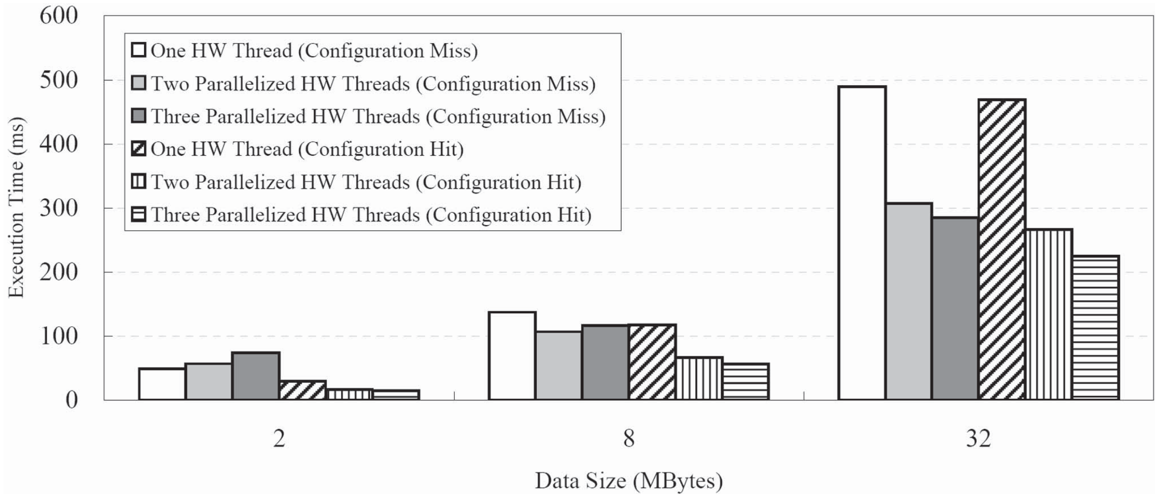


Fig. 10. Execution time of the parallelized AES encryption for different data size.

TABLE VII
THROUGHPUT AND POWER EFFICIENCY COMPARISON OF AES/DES/3DES DESIGN ON DIFFERENT PLATFORMS

Task	Platform	Clock Freq (MHz)	Power* (W)	Programming Support	Throughput (Mbps)	Power Efficiency (Mbps/W)
AES	TMS320C6201 DSP [26]	200	1.70	linear assembly code implementation	112	65.88
	Intel Pentium IV CPU [28]	2260	58.00	crypto library base API	410	7.07
	SPI Storm SP16-G160 [35]	500	7.00	SPC 1.104-O3	1120–1260	160–180
	Nvidia GeForce 8800 GTX GPU [30]	1350	177.00	CUDA with T-Box optimization	8280	46.78
	Proposed Partially Reconfigurable System	100	5.85	SPREAD base HTI and programming library	1140	194.87
DES	TMS320C6201 DSP [32]	200	1.70	linear assembly code implementation	52	30.59
	Intel Pentium IV CPU [28]	2260	58.00	crypto library base API	296	5.10
	Nvidia GeForce GTX 260 GPU [33]	1242	182.00	CUDA	4821	26.49
	Proposed Partially Reconfigurable System	100	5.74	SPREAD base HTI and programming library	698	121.60
3DES	TMS320C6201 DSP [32]	200	1.70	linear assembly code implementation	22	12.94
	Intel Pentium IV CPU [28]	2260	58.00	crypto library base API	107	1.84
	SPI Storm SP16-G160 [35]	500	7.00	SPC 1.104-O3	210–280	30–40
	Nvidia GeForce 9800 GTX GPU [34]	1688	140.00	CUDA	4500	32.14
	Proposed Partially Reconfigurable System	100	5.50	SPREAD base HTI and programming library	284	51.64

*Max TDP is used for the power consumption of TMS320C6201 DSP [27], Intel Pentium IV CPU [29] and Nvidia GeForce series GPU [31], the evaluation result from Xilinx XPower is given for our design.

memory access and the amount of row conflicts. The peak throughput of our parallelized AES implementation is 1140 Mb/s, achieving over two times speedup when compared with the single-thread implementation. The implementation makes use of three AES hardware threads running in parallel with high throughput, point-to-point streaming interconnections. The results further demonstrate the feasibility and the advantages of SPREAD in exploiting different kinds of parallelism.

VI. COMPARISON AND DISCUSSION

This section provides a comparison of the throughput and the power efficiency of different AES/DES/3DES designs on popular computing platforms, including TMS320C6201 DSP, Intel Pentium IV 2.26G CPU, SPI Storm stream processor, and Nvidia GeForce GPU.

Table VII shows that the power efficiency of the parallelized AES/DES/3DES hardware threads is superior to other computing platforms. The reasons behind these results come from the stream-oriented hardware thread interface and different kinds of parallelism exploitation.

In addition to the comparison given above, we summarize the features of SPREAD from both the architecture design and programming model points of view. Table VIII presents a comparative study on BORPH, FUSE, ReconOS, and SPREAD. The results show that SPREAD provides high

TABLE VIII
COMPARATIVE STUDY ON BORPH, FUSE, RECONOS AND SPREAD

Method	Streaming Communication Support			Programming Support	
	Communication Structure, Mechanism	Throughput	Configurable Interconnection	SW/HW Switching	Coarse Level Parallelism Exploitation
BORPH [11]	Message passing interface, General file I/O	Low	Yes	No	Pipeline Parallelism
FUSE [13]	Bus based interface, Memory mapped I/O	Medium	No	No	Not Addressed
ReconOS [14]	Bus based interface and HW FIFO, Shared memory / Message queues etc	Medium	No	No	Thread / Pipeline Parallelism
SPREAD	Streaming-based interface, Extended stream programming library	High	Yes	Yes	Thread / Data / Pipeline Parallelism

throughput streaming communication, stub thread enabled SW/HW switching, and support for exploiting multiple kinds of parallelism. These features contribute to improving hardware efficiency and runtime adaptability.

VII. CONCLUSION

This paper introduced SPREAD, a streaming-based partially reconfigurable architecture and programming model for simplifying the development of streaming applications, which demand both hardware efficiency and reconfigurable flexibility. SPREAD provided high throughput, dynamically interconnected streaming channels at structure level, as well as reconfigurable computing resource/stream/thread management at operating system level.

Through SPREAD, programmers can describe streaming applications in a unified SW/HW multithreaded model, exploiting the inherent parallelism to increase performance, while enabling SW/HW switching to improve runtime adaptability. Results from our experiments on cryptography applications demonstrated that the power efficiency offered by SPREAD is much better than the state-of-the-art GPUs. The proposed unified SW/HW interface makes programming with hardware threads easy, by removing the need to deal with low-level design details. Compared with other solutions, SPREAD can provide SW/HW switching, while achieving optimized hardware efficiency through high throughput communication and different kinds of parallelism.

Although the proposed design method was demonstrated on a Xilinx Virtex-4 FPGA, it can be applied to other FPGAs with partially reconfigurable capability. Currently, we are exploring the "one-to-many" and "many-to-one" streaming interconnections to enable a wide range of applications for SPREAD. Moreover, we plan to include necessary error checking logic within the HTI and self-healing services within the operating system kernel, to improve system reliability.

REFERENCES

[1] A. S. Zeineddini and K. Gaj, "Secure partial reconfiguration of FPGAs," in *Proc. IEEE Conf. Field-Program. Technol.*, Dec. 2005, pp. 155–162.

[2] M. Fons, F. Fons, and E. Canto, "Fingerprint image processing acceleration through run-time reconfigurable hardware," *IEEE Trans. Circuit. Syst. II, Exp. Briefs*, vol. 57, no. 12, pp. 991–995, Dec. 2010.

[3] A. Ahmad, B. Krill, A. Amira, and H. Rabah, "Efficient architectures for 3D HWT using dynamic partial reconfiguration," *EURASIP J. Syst. Archit.*, vol. 56, no. 8, pp. 305–316, Aug. 2010.

[4] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu, "Dynamically reconfigurable systolic array accelerators: A case study with extended Kalman filter and discrete wavelet transform algorithms," *IET Comput. Digital Technol.*, vol. 4, no. 2, pp. 126–142, Mar. 2010.

[5] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEEE Comput. Digital Technol.*, vol. 153, no. 3, pp. 173–180, May 2006.

[6] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: An applications perspective," in *Proc. ACM/SIGDA 9th Int. Symp. Field Program. Gate Arrays Conf.*, Feb. 2001, pp. 134–140.

[7] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. A. Najjar, and W. Bohm, "An automated process for compiling dataflow graphs into reconfigurable hardware," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 1, pp. 130–139, Feb. 2001.

[8] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. F. B. Shands, and N. Singla, "Auto-pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

[9] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek, "Stream computations organized for reconfigurable execution," *Microprocess. Microsyst.*, vol. 30, no. 6, pp. 334–354, Sep. 2006.

[10] M. Vuletic, L. Pozzi, and P. Ienne, "Virtual memory window for application-specific reconfigurable coprocessors," *IEEE Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 8, pp. 910–915, Aug. 2006.

[11] H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embedd. Comput. Syst.*, vol. 7, no. 2, pp. 1–6, Feb. 2008.

[12] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving programming model abstractions for reconfigurable computing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 1, pp. 34–44, Jan. 2008.

[13] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proc. 19th IEEE Symp. Field-Program. Custom Comput. Mach.*, May. 2011, pp. 170–177.

[14] E. Lubbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. Embedd. Comput. Syst.*, vol. 9, no. 1, pp. 1–23, Oct. 2009.

[15] *The ISO POSIX Working Group*, IEEE Standard ISO/IEC 9945, Mar. 9, 2002.

[16] M. Koester, W. Luk, J. Hagemeyer, M. Porrmann, and U. Ruckert, "Design optimizations for tiled partially reconfigurable systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 6, pp. 1048–1061, Jun. 2011.

[17] *Local Link Interface Specification*. (Jul. 2005) [Online]. Available: http://forums.xilinx.com/t5/Connectivity/Local-Link-Interface-Specification/td-p/146342.html

[18] H. Chun-Hsian, H. Pao-Ann, and S. Jih-Sheng, “Model-based platform-specific co-design methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption,” *J. Syst. Arch.*, vol. 56, no. 8, pp. 545–560, Aug. 2010.

[19] *Early Access Partial Reconfiguration User Guide*. (Sep. 2008.) [Online]. Available: http://www12.informatik.uni-erlangen.de/esmwiki/images/f/f3/Pr_flow.pdf

[20] *Xilkernl 4.0*, (Nov. 2007) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/ug758.pdf

[21] *AES Project*. (2012) [Online] Avaliable: http://www.opencore.org

[22] *DES Project*. (2012) [Online] Avaliable: http://www.opencore.org

[23] I. Gonzalez, E. Aguayo, and S. Lopez-Buedo, “Self-reconfigurable embedded systems on low-cost FPGAs,” *IEEE Micro*, vol. 27, no. 4, pp. 49–57, Aug. 2007.

[24] C. Pedraza, J. Castillo, J. I. Martinez, P. Huerta, and C. S. L. Lama, “Self-reconfigurable secure file system for embedded linux,” *IET Comput. Digital Technol.*, vol. 27, no. 4, pp. 461–470, Nov. 2008.

[25] I. Gonzalez and F. J. Gomez-Arribas, “Ciphering algorithms in microblaze-based embedded systems,” *IEE Comput. Digital Technol.*, vol. 153, no. 2, pp. 87–92, Mar. 2006.

[26] T. Wollinger, M. Wang, J. Cuajardo, and C. Paar, “How well are high-end DSPs suited for the AES algorithm?” in *Proc. 3rd AES Candidate Conf.*, Apr. 2000, pp. 94–105.

[27] *TMS320C62x/C67x Power Consumption Summary*. (Jul. 2002) [Online]. Available: http://www.ti.com/lit/an/spra486c/spra486c.pdf

[28] Z. Li, R. Iyer, S. Makineni, and L. Bhuyan, “Anatomy and performance of SSL processing,” in *Proc. IEEE Int. Symp. Perf. Anal. Syst. Softw.*, Mar. 2005, pp. 197–206.

[29] *Intel Pentium 4 Processor*. (2012) [Online]. Avaliable: http://ark.intel.com/products/27435/

[30] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” in *Proc. IEEE Int. Conf. Signal Commun.*, Nov. 2007, pp. 65–68.

[31] *NVIDIA Inc*. (2012) [Online]. Avaliable: http://www.geforce.com/Hardware/

[32] *Data Encryption Standard (DES) Implementation on the TMS320C6000*. (Nov. 2000) [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/ee3_Study_Project/DES%20Implementation%28702%29.pdf

[33] G. Agosta, A. Barenghi, F. D. Santis, and G. Pelosi, “Record setting software implementation of des using CUDA,” in *Proc. 7th Int. Conf. Inf. Technol.*, Apr. 2010, pp. 748–755.

[34] G. Liu, H. An, W. Han, G. Xu, P. Yao, M. Xu, X. Hao, and Y. Wang, “A program behavior study of block cryptography algorithms on GPGPU,” in *Proc. 4th Int. Conf. Frontier Comput. Sci. Technol.*, Dec. 2009, pp. 33–39.

[35] G. Xu, H. An, G. Liu, P. Yao, M. Xu, W. Han, X. Li, and X. Hao, “Performance and power efficiency analysis of the symmetric cryptograph on two stream processor architectures,” in *Proc. 5th Int. Conf. Intell. Inf. Hiding Multimedia Signal*, Sep. 2009, pp. 917–920.

Ying Wang received the B.S. degree from Xidian University, Xian, China, the M.S. degree from the East-China Institute of Computer Technology, Shanghai, China, and the Ph.D. degree from Fudan University, Shanghai, in 1999, 2005, and 2009, respectively, all in computer science.

She continued her research on partial reconfiguration in the State Key Laboratory of ASIC and System, Fudan University, as a Post-Doctoral Researcher, from 2009 to 2012. Her current research interests include computer architecture, the development of partially reconfigurable systems, and computing resource virtualization.

Xuegong Zhou received the B.S. and Ph.D. degrees in computing science from Fudan University, Shanghai, China, in 1989 and 2007, respectively.

He joined Fudan University in 2007, where he is currently a Research Assistant with the State Key Laboratory of ASIC and System, and the School of Microelectronics. His current research interests include logic synthesis and reconfigurable computing.

Lingli Wang (M’99) received the M.S. degree from Zhejiang University, Hangzhou, China, in 1997, and the Ph.D. degree from Edinburgh Napier University, Edinburgh, U.K., in 2001, both in electrical engineering.

He was with Altera European Technology Center for four years. In 2005, he joined Fudan University, Shanghai, China, where he is currently a Full Professor with the State Key Laboratory of ASIC and System in the School of Microelectronics. His current research interests include logic synthesis, reconfigurable computing, and quantum computing.

Jian Yan received the B.S. degree in telecommunication from Shanghai University, Shanghai, China, in 2011. He is currently pursuing the M.S. degree with the School of Microelectronics, Fudan University, Shanghai.

His current research interests include partially reconfigurable hardware accelerator customization, and system-on-chip design.

Wayne Luk (F’09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K.

He is a Professor of computer engineering with the Department of Computing, Imperial College London, London, U.K., and leads the Custom Computing Group there. He is also a Visiting Professor with Stanford University, Stanford, CA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as media processing, networking, and finance. Further information can be found in: http://cc.doc.ic.ac.hk.

Chenglian Peng received the B.S. degree in mathematics from Fudan University, Shanghai, China, in 1964.

He was a Visiting Scholar with Erlangen University, Erlangen, Germany, in 1994. He is a Professor with the School of Computer Science and Technology, Fudan University. His current research interests include computer architecture, design automation of digital systems, and fault tolerant computing.

Jiarong Tong received the B.S. degree in physics from Fudan University, Shanghai, China, in 1965.

He was a Visitor with Electronics Data Systems, Plano, TX, from 1988 to 1989. He served as a Visiting Scholar with Texas University, College Station, in 1995. He is a Professor with the Department of Microelectronic, former Dean of the Microelectronic School, Fudan University. His current research interests include reconfigurable computing, computer-aided design of integrated circuits, FPGA architecture, and digital integrated circuit design.