Journal of Systems Architecture 76 (2017) 117–132

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An energy-efficient system on a programmable chip platform for cloud applications

Xu Wang[a], Yongxin Zhu[a,f,*], Yajun Ha[b], Meikang Qiu[c], Tian Huang[a], Xueming Si[d], Jiangxing Wu[e]

[a] School of Microelectronics, Shanghai Jiao Tong University, 800 Dongchuan Road,Shanghai 200240, P.R. China
[b] Department of Electrical and Computer Engineering, National University of Singapore, 21 Lower Kent Ridge Road, Singapore
[c] Department of Computer Science, Pace University, 1 Pace Plaza, Manhattan, New York City, NY, 10038, USA
[d] Shanghai Key Laboratory of Data Science, Fudan University, 825 Zhangheng Road, Shanghai 201203, P.R. China
[e] PLA Information Engineering University, P.R. China
[f] School of Computing, National University of Singapore, Singapore

ARTICLE INFO

Article history:
Received 30 January 2016
Revised 8 October 2016
Accepted 22 November 2016
Available online 30 November 2016

Keywords:
Cloud computing
ECG classification
FPGA
Performance analysis
Reconfigurable architectures
Web server

ABSTRACT

Traditional cloud service providers build large data-centers with a huge number of connected commodity computers to meet the ever-growing demand on performance. However, the growth potential of these data-centers is limited by their corresponding energy consumption and thermal issues. Energy efficiency becomes a key issue of building large-scale cloud computing centers. To solve this issue, we propose a standalone SOPC (System on a Programmable Chip) based platform for cloud applications. We improve the energy efficiency for cloud computing platforms with two techniques. First, we propose a massive-sessions optimized TCP/IP hardware stack using a macro-pipeline architecture. It enables the hardware acceleration of pipelining execution of network packet offloading and application level data processing. This achieves higher energy efficiency while maintaining peak performance. Second, we propose a online dynamic scheduling strategy. It can reconfigure or shut down FPGA nodes according to workload variance to reduce the runtime energy consumption in a standalone SOPC based reconfigurable cluster system. Two case studies including a webserver application and a cloud based ECG (electrocardiogram) classification application are developed to validate the effectiveness of the proposed platform. Evaluation results show that our SOPC based cloud computing platform can achieve up to 418X improvement in terms of energy efficiency over commercial cloud systems.

1. Introduction

We are living in an era of cloud computing that is the backbone of embedded ubiquitous computing [1] and big data [2]. People can access powerful IT resources (e.g., computation and storage) and convenient services in the cloud via mobile devices at any time and any place. The shift of computing infrastructure from local desktop to the remote cloud improves the utilization of overall computing resources and reduces the costs associated with management of hardware and software resources for individual users.

Many services on the cloud are web-based applications, which incur a large number of concurrent requests and each of the requests involves a light-weighted task which often desires short latencies in response from energy hunger servers. For existing servers, the performance is typically limited by overheads of the network packets processing and the connection management in the NIC and OS kernel [3]. Although traditional cloud service providers such as Google and Amazon build data centers with a huge number of connected commodity computers to meet the demand of cloud computing, the ever-growing energy consumption limits the scale out of these machines due to limited energy-budget. Moreover, it comes expense related to the energy consumption and heat dissipation dominating the operating costs in such high-density computing environments. Therefore, the energy efficiency (performance per joule) becomes the key issue of building large-scale cloud computing centers [4].

Reconfigurable computing based on Field Programmable Gate Array (FPGA) technologies possesses the capability of parallel and specialized computation that can effectively exploits the task-level parallelism inherent in cloud computing with relatively low energy consumption [5]. The current trend to take advantage of FPGA in

* Corresponding author.
E-mail addresses: wang2002xu@gmail.com (X. Wang), zhuyongxin@sjtu.edu.cn, yongxin.zhu@nus.edu.sg (Y. Zhu), elehy@nus.edu.sg (Y. Ha), mqiu@pace.edu (M. Qiu), huantian@ic.sjtu.edu.cn (T. Huang), sxm@fudan.edu.cn (X. Si).

http://dx.doi.org/10.1016/j.sysarc.2016.11.009
1383-7621/

cloud computing lies in two ways: (1) to improve data transfer performance within data centers as a TCP/IP offload engine (TOE) [6]; (2) to accelerate computation-intensive applications as a hardware accelerator [7,8]. In both two methods, FPGAs are used as slave devices hosted by commodity CPUs on the master side. However, such master-slave based architecture is not suitable for web-based cloud applications, since the high communication latency between the master CPU and the slave FPGA limits the throughput of the system.

In this paper, we analyze the inefficiency of master-slave based reconfigurable architectures in dealing with highly concurrent cloud applications by proposing a performance model, and indicate that a standalone SOPC based architecture, which tightly couples network IO handling and data processing in a single FPGA, is a more promising solution in cloud computing with both high energy efficiency and high performance catering to needs of cloud applications.

Having identified a promising standalone SOPC based architecture, we then present the design and implementation of the SOPC based architecture for cloud computing in details. The major challenge to implement such SOPC based architecture is the design of a high performance TCP/IP hardware stack that is able to support high network throughput with high concurrent connections. Since third party TCP/IP offload engines are primarily optimized for inter-server data transfer among a few TCP connections in data centers, they adopt one-TCP/IP-session-per-pipeline architecture to achieve high throughput and extremely low latency. However, such TCP/IP engines are hard to scale due to resources limitation in FPGA, which is not suitable for high concurrent cloud applications. We address this problem by adopting a macro-pipeline architecture, where all TCP sessions share a centralized memory and coarse grained pipeline. Processing modules in the macro-pipeline are operated in asynchronous mode to achieve high system level throughput, while an external SRAM is used to minimize the access latency of TCP connection information that are randomly accessed and modified by different modules concurrently.

One advantage of our design by placing FPGAs on the cloud as standalone entities to provide service is its high energy efficiency at its peak performance. In other words, it supports higher performance within a fixed energy-budget constraint in data centers. Besides the efforts in hardware design, we further propose a online dynamic scheduling scheme to reduce the runtime energy consumption of the whole cluster system. By taking advantage of fast configuration ability, a load balancing node is used to shut down or power up FPGAs according to realtime workload variance in the cloud environment, saving the energy without compromising the quality of service to users.

To verify our architecture design and scheduling scheme, two case studies including a web server cluster and a cloud based ECG (electrocardiogram) classification cluster are presented to evaluate the effectiveness of proposed architecture. Design consideration and implementation details are given to show how this architecture meets the demand of cloud computing in high throughput, high concurrency, low latency, as well as low energy consumption. The major contributions of this work include:

- Performance analysis of master-slave based reconfigurable architecture and standalone SOPC based reconfigurable architecture using an analytical model.
- A massive-sessions optimized TCP/IP offload engine (described in Section 3.A) that supports up to 100K TCP sessions under 10Gbps line rate.
- An online dynamic scheduling method (described in Section 3.B) that can reconfigure or power off FPGA nodes according to workload variance to reduce the runtime energy consumption.
- Prototype of a reconfigurable cluster system based on standalone SOPC to provide cloud services, achieving up to 38X speed up in performance and 418X improvement in energy efficiency compared to the software based cloud systems.
- Prototype verification by implementing an I/O intensive web server cluster system with detailed comparison with alternate servers.
- Prototype verification by implementing a both I/O and computation intensive ECG classification cluster system, indicating support for practical cloud applications.

The rest of the paper is organized as follows. Section 2 analyzes related work briefly. Section 3 proposes a performance model to analyze the master-slave based architecture and the standalone SOPC based architecture. Section 4 describes the design details of standalone SOPC based cluster systems for cloud applications. Section 5 presents the design and implementation of two case studies including a web server cluster and a cloud based ECG classification cluster. Section 6 evaluates performance and energy efficiency of the two application cases. Our conclusions are presented in Section 7.

2. Related works

The issue of energy consumption in cloud computing has been receiving increasing attention in recent years [9]. Researchers have considered energy minimization by consolidating the workload into the minimum of physical resources and switching idle computing nodes off, with guaranteed throughput and response time. For example, Chase et al. [10] proposed an economic approach to managing shared server resources, which improves the energy efficiency of server clusters by dynamically resizing the active server set. Similar dynamic provisioning algorithms [11] are studied for long-lived TCP connections as in instant messaging and gaming. Moreover, a queuing model [12] to dynamic provisioning technique has also been studied to obtain the minimum number of servers and a combination of predictive and reactive methods has been proposed to determine when to provision these resources. Although such scheduling methods reduce energy consumption by switching idle servers to power saving modes (e.g. sleep, hibernation or power off), the long switch latency consisting of reinitialization of the system and restoring the context in commercial servers prevents their usage in real cloud systems, especially when the workloads are unstable. Recent study [13] utilises the power gating technique for FPGA-based accelerators to efficiently reduce the energy consumption of tasks running on single FPGA. However, as far as we know, there is no related work on dynamic scheduling of FPGA based cloud computing system to reduce the energy consumption.

To ensure fast changing system configuration and sharing of systems resources, Li et al. [14] proposed a rack scale composable system using PCIe switches, which is constructed as a collection of individual resources such as CPU, memory, GPU/FPGA accelerator, disks etc., and composed into workload execution units on demand. Although such composable system is a promising architecture that can improve the utilization of overall systems resources, the relative high communication latency between master CPU and slave devices (memory,GPU/FPGA) makes it less efficient in dealing with latency-bounded applications such as web-based cloud services. Compare to it, our standalone SOPC based architecture tightly couples network IO handling and data processing in a single FPGA, which significantly reduces such communication latency. Recent study [15] shows that energy consumption in network processing can be a significant percentage of total energy consumption in cloud computing. Neither high-performance nor low-power cores provide a satisfactory energy-performance trade-

off [16], since the performance of many cloud applications is typically limited by the per-packet processing overheads in the NIC and OS kernel. Therefore, there have been a few works on porting cloud workloads to FPGA with a tighter integration of the network interface. Sai et al. [16] proposed an FPGA based Memcached appliance, accelerating the low-latency access to large amounts of data for web services with 1Gbps network interface, and Michaela et al. [17] extended it to 10Gbps line rate. They utilized a similar architecture to ours with a tight proximity of network and DRAM to achieve lower latency. However, they used UDP offload engines rather than TCP offload engines to handle network communications for its minimal overhead to ensure maximum throughput over the network and the limited number of concurrent sessions in third party TCP offload engines. Compared to their work, our architecture is capable for connection-oriented cloud applications by using a novel TCP/IP offload engine that supports 100K concurrent TCP sessions.

As an attractive solution to reduce the host CPU overhead from TCP/IP processing and improve network performance, the TCP/IP offload engine (TOE) has been extensively studied both in academia and industry. Wu et al. [18] implemented a TOE acceleration hardware block and associated TCP firmware, which provided TCP/IP transmission rate up to 296 Mbps as receiver and 239 Mbps as sender. Hankook et al. [19] designed a hybrid architecture for a TCP offload engine, where the transmission and reception paths of TCP/IP are completely separated with the aid of two general embedded processors to process data transmission and reception simultaneously. A complete TOE for FPGAs without processor assist was presented in [20], which supports Gigabit Ethernet in limited hardware resources. The closest related work in TOE is [21], which supports for 10K concurrent connections in VC709 with 10 Gbps bandwidth. For each connection, this design allocates fixed-sized data buffers in the external SDRAM and maintains fixed-sized session information in on-chip block rams. Compared with it, our TOE avoids using private sending buffers for each connection, allowing sending data to be shared among multiple connections, which saves the memory space and time caused by multiple copies to private buffers. Also, we uses a event-triggered connection management in external SRAM. As a result, our TOE can support more TCP sessions (100K compared with 10K in [21]) with similar performance and resource consumption. Recently, solutions from industry have already extended the throughput of TOE to 10Gbps–40Gbps [22,23]. Up until 2014, all of the available TOEs were ultra-low latency and low session count (below 256 concurrent sessions), with the key driving applications being high-frequency trading and inter-server data transfer in data centers. As a result, these third party solutions typically support one TCP/IP session per instantiated TOE, which limits the number of concurrent TCP sessions due to the FPGA resources. In 2014, Intilop has announced a high session count variant supporting up to 16,000 sessions. However, as Intilop has not disclosed the design, it is hard for us to make a detailed architectural comparison at this stage.

3. Performance analysis of two reconfigurable architectures

3.1. Master-slave based architecture

Many reconfigurable systems using FPGA are built as the master-slave systems [8,24,25]. In each computation node, a general purpose processor based computer works as master, which runs operating system and handles communications and data transfer with the slave systems. Slave systems consist of I/O devices including hard drives and network interfaces, as well as reconfigurable logic based on FPGAs. Communications between master side and slave side are delivered via high speed onboard interconnections such as PCI-E.

According to Amdahl's law [26], an arbitrary algorithm can be divided into portions that can be made parallel and portions that cannot be parallelized, i.e., to remain serial, and the speedup of an algorithm in parallel computing is limited by the time needed for the sequential fraction of the algorithm. Modern reconfigurable devices can provide abundant computing resources to accelerate the parallel portion of an algorithm. However, performance speedup may not be always achieved due to with many long latency control communications and I/O operations, thus increasing the sequential portion.

To reveal such architectural limitations in parallel implementation of an application in reconfigurable systems, we further divide sequential portion into three parts: serial computation (S), control I/O operation (CIO), and data I/O operation (DIO), based on which we define the time consumed by each part as T_s, T_{cio}, and T_{dio} respectively. Also we define T_p as the time spent on parallel computation, then the overall executing time T in a complete synchronous system is given by:

$$T = T_s + T_p + T_{cio} + T_{dio}. \tag{1}$$

Considering optimization techniques like overlapping I/O and processing, we can also define the overall executing time T in a complete asynchronous system as:

$$T = max\{T_s, T_p, (T_{cio} + T_{dio})\}. \tag{2}$$

However, real reconfigure systems are falls somewhere in-between of the Eq. (1) and Eq. (2). For simplicity, we use Eq. (2) to analyze the two reconfigure systems.

Since the serial computation portion of an application is in most situations executed on a processor, we can calculate the T_s as:

$$T_s = \frac{S}{f_s \times \sum_i \frac{N_i}{CPI_i}}, \tag{3}$$

where S represents the workload of serial computation portion which is defined by number of instructions. N_i is the number of instructions that can be issued simultaneously of functional unit type i, CPI_i is the average number of cycles per instruction (such as DSP, ALU) for functional unit type i and f_s is the operating frequency of the device. Since the S is defined that cannot be parallelized, we simplify Eq. (3) as:

$$T_s = \frac{S \times CPI}{f_s}. \tag{4}$$

In reconfigurable systems, the parallel computation part of an application is most likely executed by hardware logic in FPGAs, so the value of T_p can be calculated as:

$$T_p = \frac{P}{R \times \frac{f_p}{CPP}}, \tag{5}$$

where P represents the workload of parallel computation portion which is defined by number of operations. R is the number of reconfigurable resources available for parallel computation. For example, the total number of Int32 IP cores of adders and multipliers that an FPGA can support. CPP is the average number of clock cycles per operation per reconfigurable resource and f_p is the operating frequency.

The process of control involves a sequence of message passing operations between master CPU and slave devices. Although modern high speed interconnections are able to provide an extremely high throughput for data transmission, it brings a relatively high latency. Since such control messages always contain little volume of payload and a control process includes several times of message passing, it makes the communication delay the dominant factor in T_{cio}:

$$T_{cio} = N_c \times D_c. \tag{6}$$

Eq. (6) gives the definition of T_{cio}, where N_c is the number of communications for control operation and D_c is the delay for such communications.

We assume data I/O operations are all large data transmissions and small data transmission can be referred to control operations. The throughput of I/O interface rather than communication latency becomes the major factor. Since it needs more than one data copy operations via different I/O interface to feed data to the execution unit, the time consumed by I/O operations can be defined as:

$$T_{dio} = \sum_j \frac{\Phi}{\theta_j}, \tag{7}$$

where Φ is the size of data to be transferred and θ_j is the max throughput of each interface j where data copy/transfer operation occurs.

To sum up, Eq. (2) is specified as:

$$T = max\left\{\frac{S \times CPI}{f_s}, \frac{P}{R \times \frac{f_p}{CPP}}, \left(N_c \times D_c + \sum_j \frac{\Phi}{\theta_j}\right)\right\}. \tag{8}$$

With the definition given by Eq. (8) and considering the energy consumption of the whole system including both general purpose processors and FPGAs, we can infer that the applications that perfectly match the master-slave reconfigurable system thus achieving high energy efficiency must have following properties: 1) a comparative portion of serial computation workload that can keep the general purpose processors in high utilization; 2) less control interactions between master and slaves; 3) the data from I/O interface must satisfy the needs of parallel processing in FPGAs.

However, many services on the cloud are web-based applications, which include a large number of concurrent tasks and each of the tasks is light-weighted. In other words, these applications contain very little portion of serial computation-intensive workload but rather large part of parallel computation coupled with complex control and I/O operations. In this case, the master-slave based reconfigurable systems become inefficient. Since D_c is large in master-slave systems, when N_c increases to a certain threshold, the time spent in message interaction between host and FPGA devices may consume much more time than that for parallel computation in FPGAs. Also, when there is a bottleneck in the chain of transferring data from I/O to FPGA or large I/O delays caused by too many data copy operations, poor performance would be the consequence because of the insufficiently data supply for parallel computational cores.

3.2. SOPC based architecture

The SOPC based reconfigurable architecture integrates the functionality provided by master side in a master-slave system into FPGAs. General purpose processors with high performance and high energy consumption are replaced by embedded microprocessors, while off-chip interconnect are replaced by on-chip interconnect with low latency. I/O facilities are directly connected to FPGAs where controller circuits of such I/O interface are implemented as IP blocks. As a result, it is more suitable for cloud based applications for the following reasons:

1. It narrows the distance between control unit and computational unit by using direct message passing interface such as FIFOs or shared memories, so that it decreases the control latency (D_c).
2. Data from I/O interfaces are directly fed to computational logic via raw interface with extremely high throughput. As such, the architecture decreases the number of data copy operations and eliminates the data transmission bottleneck that may exist in the master-slave architecture. Also, direct I/O access with hardware control logic can offload some of control workload that should be done by processors. As a result, it reduces *Nc*.

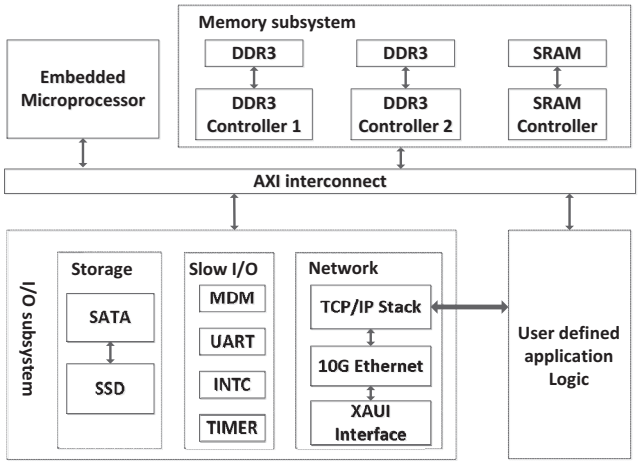


Fig. 1. SOPC based architecture for Cloud applications in single FPGA.

3. Using embedded microprocessors instead of general purpose processors in applications to incur little serial computational workload can dramatically reduce the overall energy consumption of the system, while it still provides identical functionalities with acceptable performance.

4. Design details of SOPC based reconfigurable clusters for cloud applications

The inherent nature of the cloud computing imposes more demand on cloud-side system for high throughput network I/O, high performance computing capability and fast storage. In order to meet the demand in a standalone FPGA, we designed a standalone SOPC based architecture for cloud service that tightly couples network IO handling and data processing in a single FPGA, which is illustrated in Fig. 1. It includes an I/O subsystem, a memory subsystem, an embedded microprocessor and a user defined application logic. In the I/O subsystem, network protocols from physical layer to transport layer are integrated in hardware pipeline to achieve high throughput and low latency. It supports 10Gb optical ethernet via XAUI interface as well as the entire TCP/IP stack. We also integrate a SATA controller to support non-volatile mass storage devices. In the memory subsystem, we use two types of external memory for different purposes: Two DDR3 SDRAMs, which provide 16GB memory space in total, are used as data memory to store bulk data such as upstream and downstream data, for its high storage density and high burst throughput; One 72Mb SRAM is used to store data structures of TCP sessions for its low access latency. Moreover, an embedded microprocessor is involved to incorporate with AMBA AXI4 interconnects to provide 3 major functions as follows: (1) Initializing the whole system by configuring registers of system modules and peripherals; (2) Providing an AHCI based driver for SATA controller; (3) Maintaining a file system based on SSD. Unlike the former-described common component, the user defined application logic is unique for different cloud applications, which implements the application level algorithm in cloud service in specialized hardware pipelines.

4.1. Massive-sessions optimized TCP/IP stack

In the scenario of cloud applications, the major disadvantage of third party TCP/IP offload engines that would outweigh advantages is that they are primarily optimized for massive data volume transmission rather than number of concurrent connections to boast their support for bandwidth of 10+Gbps, where each TOE

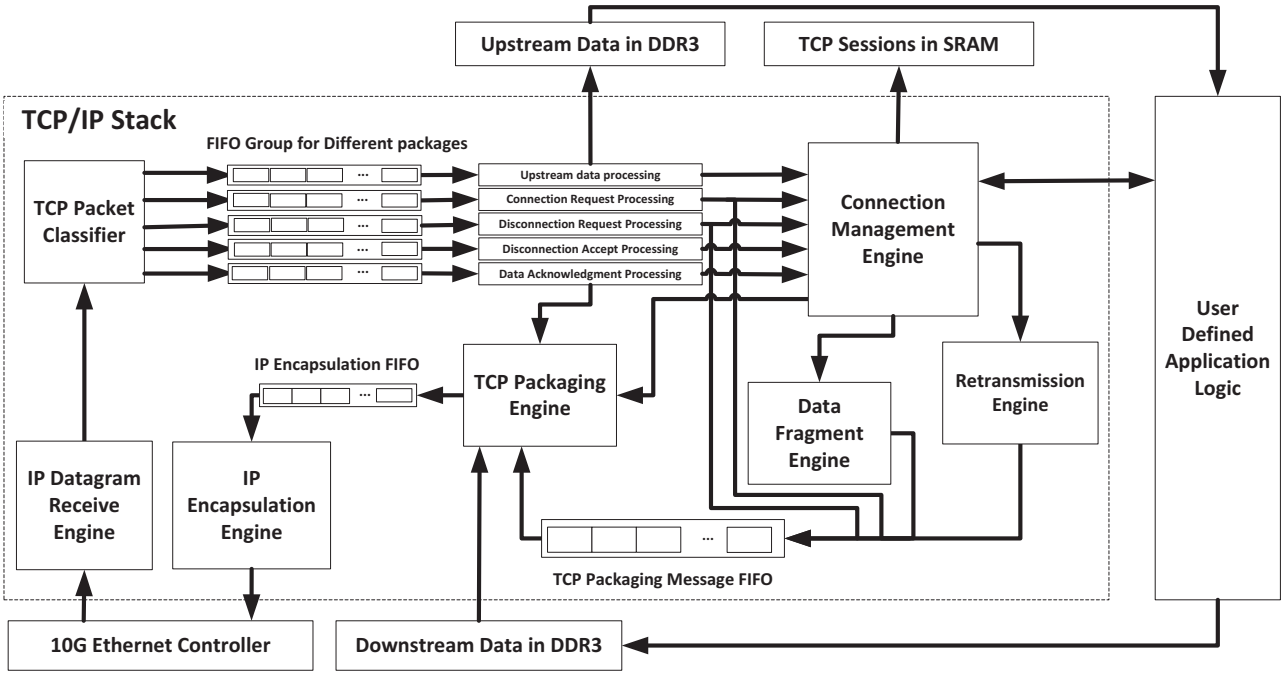


Fig. 2. Mircro-architecture of TCP/IP stack.

only handles no more than hundreds of TCP connections concurrently. As a result, they adopt one TCP/IP session per instantiated pipeline architecture [22,23], where each session includes full TCP function as well as private TX and RX buffer using on-chip BRAMs. Multi-sessions can be supported by instantiating multiple copies of such pipeline. However, web based cloud computing is quite a different application scenario in that the TOE needs to maintain more than ten thousands of concurrent connections where each connection only contributes a little portion of total throughput. Thus these third party TOEs no longer satisfy such requirement due to limited on-chip resource to support more TCP sessions.To address the problem, we propose a massive-sessions optimized TOE. We use a macro-pipeline architecture, where all TCP sessions share a coarse-grained pipeline with carefully organized TX, RX buffer. Also,we develop a hardware based connection management to support rapid scheduling of up to 100K TCP sessions. We show our architecture in Fig. 2 with the detailed explanations given in the following subsections.

4.1.1. Macro-pipeline architecture

In order to achieve high system throughput, all processing modules in the macro-pipeline are operated in asynchronous way, each of which handles a portion of processing stage or directly responds to a specific type of TCP packets. These processing modules are event triggered, that is they continuously read the events or messages from the input FIFO, process the events, and send messages to the next stage without any synchronizations. Specifically, for incoming data processing, data from network are first offloaded from 10G Ethernet and unpacked by IP datagram receive engine to obtain TCP packets. Then these TCP packets are divided into different categories based on the control bit in their TCP headers and delivered to different processing engines respectively. Each processing engine deals with a particular type of TCP packet such as SYN packet, FIN packet or ACK packet and directly generates feedback packets to response. The connection management engine deals with the connection and disconnection of TCP links and schedules of the data transmission order among all established links. For outgoing data processing, the data fragment engine segments data

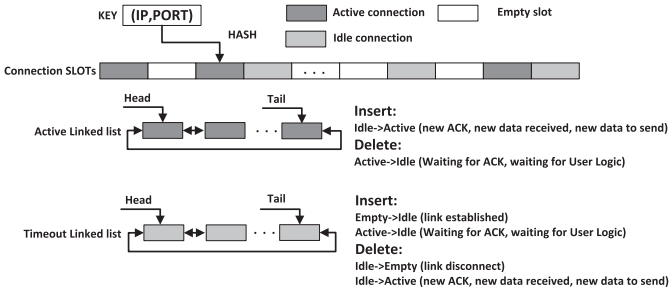


Fig. 3. Memory allocation and scheduling of TCP connections.

that is to be transmitted into small data packages. Then these data packages are encapsulated with TCP header and IP header successively by TCP packaging engine and IP encapsulation engine. When a timeout occurs, the retransmission engine retransmits the lost TCP packets based on the information from the most recent data ACK number.

4.1.2. Connection management

Unlike conventional TOEs that support only no more than hundreds of TCP sessions, to support more than ten thousands of TCP sessions in our TOE is no trivial task. Due to the limited on-chip memory resources, connection information can only be stored in external memory, while the shared memory may become the bottleneck of the system when it incurs random accesses from different processing modules concurrently. We mitigate the impact of the problem in two ways: to reduce access latency and to reduce the number of memory accesses. We first choose SRAM instead of DRAM to store TCP sessions for its much lower access latency. Then, as described in Fig. 3, the memory space of external SRAM is segmented into a number of consecutive TCP connection slots, where we use a hash mechanism with source IP address and port number as the key to fast lookup of a dedicated TCP connection. We avoid using round-robin scheduling scheme, which is widely used in TOEs that support small number of TCP sessions, to sched-

ule the execution of data transmission among established TCP connections. Because in web-based cloud computing scenario with a large number of concurrent connections, it is time consuming to check every connection slot to find an active connection that is able to sent data. Instead, we maintain two linked list to fast locate active connections and timeout connections. The active linked list is used to store TCP connections that have an event to be processed. The connection management engine continuously obtains the connection in the head, executes the event, then puts the connection to the tail of timeout linked list. When an event (new ACK received, new data received or new data to send) for certain connection occurs, the connection is inserted to the tail of active linked list. On the other hand, the timeout linked list is used for monitoring the transmission timeout of idle connections. Idle connections are inserted to the tail of timeout linked list with a timeout point that adds a fix timeout value to current time. Since the timeout linked list is naturally sorted in timeout point, the retransmission engine only need to check through the head of the list to obtain the most timeout connections, avoiding recurrently traversing the whole established TCP connection slots that may impose access burdens to the shared memory.

4.1.3. TX, RX buffer allocation

Due to limited on-chip resources, it is impossible to offer each TCP session a private TX and RX buffer using BRAMs. Instead, we directly store the data in DDR3 external memory. However, the organization of upstream data and downstream data for each TCP connections is different. For upstream data, we maintain a buffer pool (a continuous memory space in DDR3 SDRAM) that is divided equally into segments. The size of each segment can be programed according to the need of application scenario. A buffer allocation stack is used to manage the allocation and reclaim of the buffer resources. When a new connection is established, the connection management engine acquires a new buffer from the buffer pool, and registers it in the connection information. Then upstream data can be offloaded to the buffer, where it can be directly accessed by user defined logic. During TCP disconnection, the buffer is recycled back to the buffer pool. For downstream data, however, our TCP stack does not assign a specific TX buffer to each established TCP connection. Rather we leave user defined logic to organize the data to be sent. Specifically, the user defined logic can trigger the TCP stack to send data by registering one or more data ready events, each of which contains the starting address and the length of prepared data in DDR3 SDRAM. This leads to a series of benefits for performance: (1) The user defined logic needs not to wait in case that the send buffer is full; (2) It eliminates the data copy operations from application level to transport level; (3) It supports multiple data transmission without the involvement of user defined logic; (4) Many TCP connections can share the same data to be sent (e.g. users' requests for the same webpage in webserver application), which saves the memory space and time caused by multiple copies to private send buffers.

4.2. Online dynamic scheduling

Data centers are typically designed to provide processing capacity for potential peak workload. However, actual request load imposed to data centers changes over time. It is a waste of energy to activate all its processing power when system is at a low load. Fortunately, the standalone SOPC based architecture where every FPGA works as an independent system offers high flexibility in scheduling, which can be used to address the problem. Since the configuration and shut down time of a FPGA is negligible compared to that of a general-purpose computer, it is possible to adjust the number of activated computing nodes according to realtime workload by directly powering up or down FPGAs in the cloud environment to save the energy, while the latter must keep awake most of the machines.

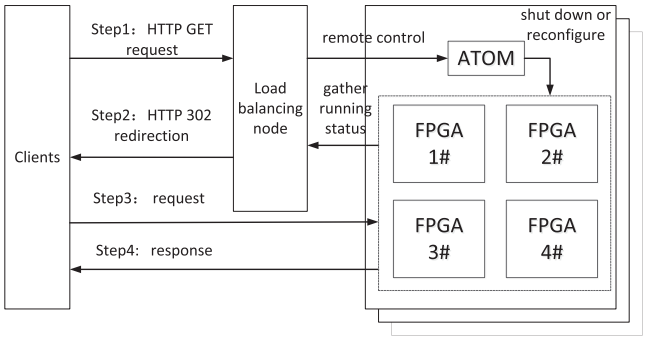


Fig. 4. Online dynamic scheduling in standalone SOPC based reconfigurable clusters.

Due to our impressively short reconfiguration time in seconds (less than two seconds) rather than minutes, which most of FPGA users take to reconfigure FPGA via JTAG cables, as we store reconfiguration bits in NOR flash connected to FPGA via BPI interface, we can exploit the advantages of a lot of dynamic load balancing methods on web-based systems [27], many of which can be adopted in our architecture. We chose HTTP redirection technique to route incoming requests into our FPGA based cloud servers. Fig. 4 demonstrates the process of such dynamic scheduling. First of all, the user on the client side connects and sends an HTTP GET request to the load balancing node. The load balancing node picks up a running FPGA that has the least workload and sends its IP back to the client side as an HTTP 302 redirection response. Then, the user sends the request to the allocated FPGA to receive services on the cloud side.

The load balancing node periodically (five second interval for instance) gathers running status of each FPGA to record the workload distribution of the whole system. Note that adding communication support between the load balancing node and the FPGA would cause additional chip area usage. However, in our system, we can reuse the network data path we have already build to support online dynamic scheduling on the FPGA side. The polling action is simply done by sending a unique HTTP GET request to each FPGA, which is been processed by our TCP/IP stack. Then the on-chip microprocessor will collect data of running status and sent them back as the HTTP response. All that are needed are several registers to store the running status and a small piece of code running on the on-chip microprocessor to deal with the unique HTTP GET request. For different applications, running status and scheduling algorithm can be various. In our implementation, the average system throughput during the last time interval of each FPGA is obtained to monitor the condition of loading. We define T_i as the current throughput for $FPGA_i$, T_m as the maximum throughtput that a single FPGA can afford to provide services under certain QoS (Quality of Service) requirement, and N is the current number of total active FPGAs. So the system level load rate is defined as:

$$L_{current} = \frac{\sum_{i=1}^{N} T_i}{N \cdot T_m} \tag{9}$$

Common practice (defined as basic algorithm) uses an upper bound L_u and a lower bound L_l of system level load rate to decide whether to increase or decrease the number of the activated FPGAs. When the system load rate $L_{current}$ drops below the lower bound L_l, the load balancing node first stops routing the user requests to the FPGA that contains the least workload, and informs the onboard configuration module to power off such FPGA when it becomes idle. On the contrary, when the system load rate $L_{current}$

rises over the upper bound L_u, the load balancing node informs the onboard configuration module to power up and configure a resting FPGA to provide service.

However, the parameter setting of the scheduling upper bound L_u and lower bound L_l is no trivial. The higher L_u increases the possibility of service failure, while the lower L_u leads to the waste of performance. Also, the higher L_l may cause the oscillation between wake up and power off and the lower L_l gives rise to the waste of energy. To address the problem, we introduced the following two optimizations to the basic algorithm:

4.2.1. Predictive scheduling

For every scheduling point, we first predict the system load of the next scheduling period (L_{next}) by quadratic fitting the system load data of the last three samples. Specifically, we define $S^{(-j)} = \sum_{i \in ActiveList^{(-j)}} T_i^{(-j)}$ as the total throughtput of all FPGAs at the last j scheduling point, where $S^{(0)}$ stands for current total throughput. Then, we can compute the system load of the next scheduling period as $L_{next} = \frac{S^{(-2)} - 3S^{(-1)} + 3S^{(0)}}{|ActiveList| \cdot T_m}$. We use the maximum value of L_{next} and $L_{current}$ to decide whether to activate more FPGAs. Compared to the basic algorithm that tends to choose a moderate L_u to avoid service failure when the system encounters explosive growth of load, our optimization can predict the trend of load variation and make adjustment in advance. As a result, we can use a higher L_u to make the whole system more energy efficient without worrying service failure.

4.2.2. Sliding window

Since the overall loads on the cloud side are sum of massive individuals requests, it tends to be smoothly changed and stable with small fluctuation in most of time. Therefore, it is important to improve the energy efficiency when the system is under such stable state. However, the basic algorithm is hard to guarantee a high energy efficiency during the stable period, which tends to choose a low L_l to avoid oscillation between wake up and power off. For example, If the L_l is set to 50% and L_u to 80%, while the system load fluctuate around 60% ± 8%, the basic algorithm would not decrease the number of activated FPGAs to improve the system utilization. To address the problem, we setup a sliding window of size k, and keep track of the maximum value of the whole system throughput ($S_{max} = max\{S^{(0)}, S^{(-1)}, \ldots, S^{(-k+1)}\}$) among the last k samples. By assuming that there is a high probability that the load during the next scheduling period is no more than S_{max}, we can reduce the number of activated FPGAs to make system in highest utilization under the upper bound L_u, which is calculated as $\lceil \frac{S_{max}}{T_m \cdot L_u} \rceil$. Although such optimization increases the response latency of powering off FPGAs when system load rate $L_{current}$ drops, it significantly improves the energy efficiency when the system is under stable state.

More specifically, we describe our algorithm in pseudo-code as shown in Algorithm 1.

Algorithm 1 Online dynamic scheduling.

Input: *ActiveList* ▷ set of activated FPGAs
Input: *IdleList* ▷ set of ready-to-power-off FPGAs
Input: *PowerOffList* ▷ set of power-off FPGAs
Input: *interval* ▷ scheduling interval
Input: k ▷ window size
Input: L_u ▷ scheduling upper bound

loop
  for all FPGA i in $ActiveList \bigcap IdleList$ **do**
    gather current throughput T_i
  end for
  $S^{(0)} \leftarrow \sum_{i \in ActiveList} T_i$
  $L_{current} \leftarrow \frac{S^{(0)}}{|ActiveList| \cdot T_m}$
  $L_{next} \leftarrow \frac{S^{(-2)} - 3S^{(-1)} + 3S^{(0)}}{|ActiveList| \cdot T_m}$ ▷ quadratic fitting
  $L_{target} \leftarrow \max\{L_{current}, L_{next}\}$
  if $L_{target} > L_u$ **then** ▷ activate new FPGAs
    repeat
      if *IdleList* is not empty **then**
        $node \leftarrow IdleList$.maxT_pop()
        $ActiveList$.push($node$)
      else if *PowerOffList* is not empty **then**
        $node \leftarrow PowerOffList$.pop()
        reconfigure $FPGA_{node}$
        $ActiveList$.push($node$)
      end if
      $L_{target} \leftarrow \frac{|ActiveList|}{|ActiveList|+1} L_{target}$
    until $L_{target} < L_u$
  else
    $S_{max} \leftarrow \max\{S^{(0)}, S^{(-1)}, \ldots, S^{(-k+1)}\}$
    $N_t \leftarrow \lceil \frac{S_{max}}{T_m \cdot L_u} \rceil$
    while $|ActiveList| > N_t$ **do** ▷ deactivate FPGAs
      $node \leftarrow ActiveList$.minT_pop()
      $IdleList$.push($node$)
    end while
  end if
  for all FPGA i in *IdleList* **do** ▷ poweroff FPGAs
    if $T_i == 0$ **then**
      $IdleList$.delete(i)
      power off $FPGA_i$
      $PowerOffList$.push(i)
    end if
  end for
  wait($interval$)
end loop

5. Case studies

In this section, we use two case studies to verify the effectiveness of our design and implementation of cloud based applications using the proposed architecture. The web server application, which plays a fundamental role in current cloud computing infrastructure, is chosen as the base case study. Since the web server is the major channel to connect users and cloud services, we use this case study to evaluate how this architecture meets the requirement of low delay, high bandwidth and high concurrency I/O performance in cloud environment. The second case study is a cloud based ECG classification which is an important bio-cloud application [28]. Unlike webserver application which is only I/O-intensive, the cloud based ECG classification is an application that both I/O-intensive and computation-intensive. We use this case study to demonstrate how this architecture handles realistic cloud-based workloads. Since we have already presented the common module of the SOPC Based Architecture in details, in this section we mainly focus on the design of the user defined application logic.

5.1. Webserver

The primary function of a web server is to deliver web pages or files to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP). A user agent, commonly a web browser, initiates communication by making a request for a specific resource via HTTP and the server responds with the content of that resource. In our design (Fig. 5), the user defined application logic contains two main modules: a URL parser and a data locating module. To narrow the bandwidth gap between network I/O and SSD storage I/O, a cache is maintained in

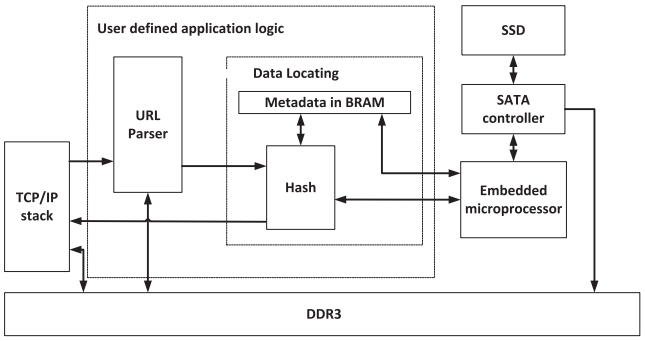


Fig. 5. Architecture of the user defined application logic for webserver.

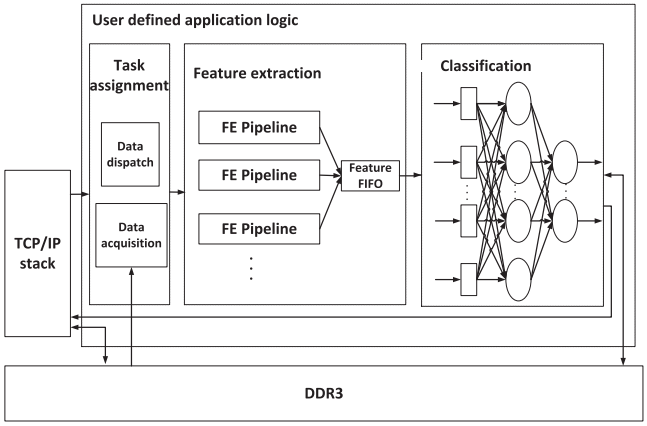


Fig. 6. Architecture of the user defined application logic for cloud based ECG classification.

the DDR3 memory. For simplicity, we use embedded microprocessor to assist the cache management while a hash table in on-chip block ram is used to fast locate the cached address of request file by hardware. When TCP/IP stack offloads the upstream data, the URL parser reads the HTTP GET request from the DDR3 memory and parse it to obtain a file name. Then the data locating module checks if the requested file is cached in DDR3 memory using the hash table. If yes, the data locating module directly signals the data fragment engine by writing the address and the length information of such cached data to the corresponding data send message without copying the data to a buffer; Otherwise, it notifies the embedded microprocessor to move the data from the SSD to the cache and then repeats the previous operations. Due to limited space of the paper, we do not extend further with detailed hardware implementation of the two modules, which is not the core issue of this case.

5.2. Cloud based ECG classification

The electrocardiogram (ECG) is a graphic representation of the electrical activity of the heart, which is a useful tool to diagnose heart diseases. Since the heart disease is still the most common cause of death in the modern world, the early diagnosis of such disease becomes significantly important. Recent advances in mobile technology and cloud computing have inspired numerous designs of cloud-based health care services and devices [28,29]. Medical data such as ECG can be collected and transmitted automatically to the cloud based analysis system from anywhere and timely feedback can be returned to patients through the network. The entire process of automated diagnosis can be generally subdivided into three procedures: beat detection, feature extraction, and classification [30].

Beat Detection: The QRS complex detection algorithm described in [31] is used for our design of beat detection. This algorithm is based on wavelet transform (WT), in which a target signal is decomposed into an approximate signal and a set of relatively high-frequency wavelets. Each stage of WT consists of a high-pass filter $G[\cdot]$, a low-pass filter $H[\cdot]$ and two down-samplers by 2 ($\downarrow 2$). The down-sampled outputs of first high and low-pass filters provide the detail, $D1$ and the approximation, $A1$, respectively. Following the first level of decomposition, the approximation $A1$ is further decomposed in the second level using the same filters, and so on.

The detection of QRS complex is mainly focused on the detection of R peaks in the raw ECG data. In D2, every special wave where a negative minimum point is followed by a positive maximum point corresponds to each R peak. In other words, we can capture the position of R peaks by identifying the zero-crossing of extremum pairs in D2. Then the QRS complex is extracted from the raw ECG data by segmenting a fixed number of samples centered at R peaks.

Feature Extraction: We use the method presented in [32] to extract features from QRS complex. The two-level discrete wavelet decomposition produces signal components in different subbands. In order to reduce the dimensionality of the extracted signal components, 11 statistics over the set of the wavelet coefficients were used to represent the time-frequency distribution of the ECG signals:

1. Variance of the original QRS complex signal;
2. Variance of the wavelet coefficients in each subband (D1,D2,A2);
3. Variance of autocorrelation functions of the wavelet coefficients in each subband (D1,D2,A2);
4. Ratio of the minimum to the maximum of the wavelet coefficients in each subband (D1,D2,A2);
5. instantaneous RR intervals.

Classification: We use artificial neural network (ANN) to classify the extracted features, which is a widely used method for classification in machine learning and pattern recognition.

The architecture of our cloud based ECG classification in FPGA is showed in Fig. 6. While users connect to the FPGA and upload their ECG data through TCP/IP protocol, the TCP/IP stack engine handles these TCP connections and offloads the ECG data to the DDR3 memory. Then the TCP/IP stack engine sends the connection information messages to the task assignment module through a FIFO. The latter periodically polls all feature extraction (FE) pipelines to find an idle one, and then reads and dispatches the ECG data associated with head-of-the-FIFO TCP connection to the idle FE pipeline. Since the feature extraction stage is the most computation-intensive and time consuming stage, we put multiple FE pipelines to meet the requirement of system level throughput. The input of each FE pipelines is the sequential samples of ECG data related to one user. After several processing steps including R-peak detection, 2 level wavelet transformation, autocorrelation computation, variance computation, and min/max computation, FE pipeline outputs 11 statistics features described above as well as its connection information message to a feature FIFO. The classification module contains a pre-trained ANN with 11 input neurons, 20 hidden neurons and 6 output neurons. It automatically reads features from the feature FIFO, classifies the type of the ECG data, writes the result to the DDR3 memory, and then notifies the TCP/IP engine to send the diagnostic result back to the user.

Since both the R peak detection and the feature extraction contain a wavelet transformation procedure, we share a two level WT module between these two processes. The architecture of the fea-

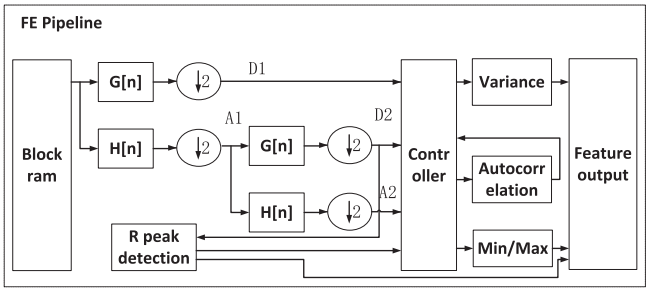


Fig. 7. Architecture of feature extraction pipeline.

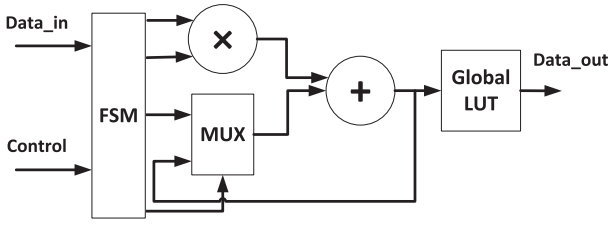


Fig. 8. Architecture of light-weight neuron design in FPGA.

ture extraction pipeline is showed in Fig. 7, when ECG data is ready in the local block ram, the WT module reads it out and executes a pipelined wavelet transformation. The D2 output is sent to R peak detection module, where all R peak positions as well as the 64-point QRS segments centered at R-peaks are extracted from the record. The latter records the position of the R peak and Also, the instantaneous RR intervals are computed and sent to the feature output module. After that, a second wavelet transformation is executed to the extracted QRS segments. The controller stores all subband data and controls the execution order to compute all statistics feature via different modules including Min/Max module, autocorrelation module, and variance module. In our design, the memory space in the block ram is divided into two identical parts to support ping pong operations, which can hide the latency related to data acquisition.

The architecture of single neuron in hidden layer and output layer is illustrated as Fig. 8. Since the most time consuming stage is feature extraction stage rather than classification stage, we adopt a more light-weight neuron design to save the on-chip resources. Each neuron contains a finite state machine to control the access of data from the output of the former neurons. During each cycle, the neuron executes one multiply-accumulate operation to one of its input data with its pre-trained weight. The implementation of the sigmoid function use a look-up table (LUT) stored in the on-chip block Ram [33]. The use of the LUT reduces the resource requirement and improves the speed. In our design, a global 64K depth and 16bit width LUT is used to compromise between the resources utilization and the resolution of the output.

6. Experimental results and discussion

6.1. Building experimental prototype platform

We have been involved in building mimicry computer [34], and used this reconfigurable platform (Fig. 9) to evaluate the our SOPC based architecture for cloud applications. The current prototype platform consists of 16 computing nodes, each of which is packaged in a standard 2U rack-mountable, as shown in Fig. 10. Each computing node houses one mother board and four daughter boards. Each daughter board contains: 1) a XC6VLX550T FPGA from Xilinx; 2) three SO-DIMM channels that support up to 16GB

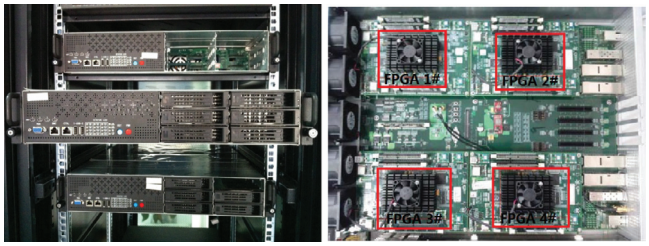


Fig. 9. Experimental prototype platform. (a) Reconfigurable prototype platform containing 16 2U rack-mountable computing node; (b) one of its quad-FPGA reconfigurable computing node.

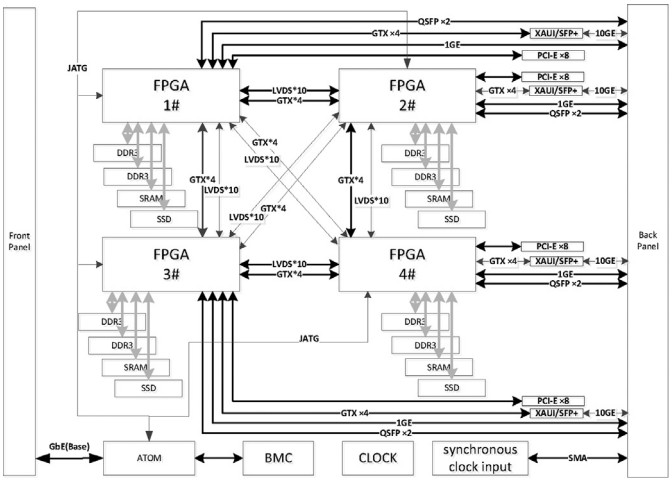


Fig. 10. Board architecture for single node in experimental prototype platform.

DDR3 SDRAM and 72Mb SRAM; 3) 10G Ethernet with XAUI/SFP+ interface; 4) SATA3 interface connecting an Intel 320GB SSD; 5) 32MB NOR flash in BPI interface; 6)PCI-E X8 interface. Data transfer between adjacent FPGAs can be carried out directly through point to point interconnections in mother board. All FPGAs are interconnected by a fiber channel switch from Cisco with their IP addressed pre-defined in the flash that is attached to each FPGA.

For the purpose of remote configuration, an onboard configuration module based on an Intel Atom CPU is integrated on each mother board. Tools are provided for users to manipulate all kinds of operation such as starting up or shutting down certain FPGA, downloading the bitstream, even debugging remotely using Chipscope, when a PC host is connected to the Atom system through network. By using BPI configuration interface, it only takes less than two second to configure the FPGA.

6.2. Resources utilization

As mentioned above, we implemented both two applications on our reconfigurable prototype platform. Each instance of the SOPC based architecture was implemented in a Xilinx XC6VLX550T FPGA. The embedded microprocessor is implemented by Microblaze, a light-weight soft core provided by Xilinx. The system clock frequency of both two implementations are 156.25Mhz which is equal to the frequency of 10Gbps PHY interface. Table 1 shows the device utilization summary of main modules for single core, which is generated by Xilinx ISE 14.1. All common modules in total use only 6.4% of total slice registers, 13.5% of total slice LUTs, 25.1% of total BRAMs and less than 1% of total DSP48E1s, which leaves most of on-chip resources for user defined application logic. Note that

Table 1
Device utilization summary for single core.

XC6VLX550T				
	Slice Reg	LUTs	BRAM	DSP48
Available	687,360	343,680	2528	864
Common Modules				
TCP/IP Stack	21,312	22,604	126	6
SATA Controller	3108	4816	10	0
Microblaze	1620	2185	498	3
DDR3 Controller	10,028	8200	0	0
AXI interconnect	4155	4989	0	0
Ethernet	3704	3555	0	0
User Defined Application Logic				
web server	9727	11,563	114	0
ECG classification	87,242	72,972	162	450
The-state-of-the-art TOE in academia				
TOE[21]	20,611	19,026	279	

our TCP/IP stack has a similar source usage as the-state-of-the-art [21] TOE in academia, but supports more concurrent TCP sessions.

6.3. Performance evaluation of the SOPC based web server

The performance of the SOPC based web server was evaluated by Spirent Avalanche 3100 [35], a popular network performance testing equipment that is able to evaluate OSI Layer 4–7 stateful traffic performance over multi-10 Gbps physical layers. Three performance metrics are measured including:

- Throughput, measured in bits per second (bps), which is the rate of successful data delivery in a given time period;
- Number of HTTP Transactions per second, which refers to the number of complete HTTP transaction including connection establishment, file transmission, and connection disconnection, performed by web server per second. This metric is highly related to the response time of single transaction, when requested files are very small;
- Max number of concurrent connections, which refers to the max number of active TCP links processed concurrently.

Results were compared with other two main stream industrial systems:

- Apache HTTP web server [36], which is the most widely used open-source HTTP server in industry. We launched Apache 2.2 HTTP web server instance on IBM x3635 M3 servers with Intel X5650 CPU (2.66 GHz), 8GB DDR3 memory and Intel X520-SR2 10GE network adapter.
- Master-slave based web server where we connected our daughter board as a slave device to IBM x3635 M3 servers via PCI-E interface. We integrated the QuickTCP [22] IP core, a third party 10GE TCP/IP stack in FPGA to offload network data processing, leaving the master side CPU handling application level functions such as file management and look up.

Files of different sizes were stored in the SSD for SOPC based web server, while the master-slave based implementation system held such files in the hard drive hosted on the master side.

6.3.1. Throughput

Fig. 11 shows that the peak throughput of SOPC based web server is 8.3Gbps which is slightly lower than Apache and master-slave based web server. However, when the size of requested file became small, the standalone SOPC based web server performed much better than other two systems. Because in this situation, the processing delay of single complete HTTP transaction including the setting up and tearing down of TCP connection turned out to be

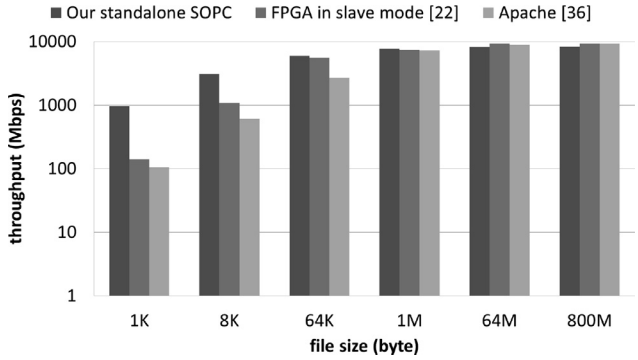


Fig. 11. Throughput at different file sizes.

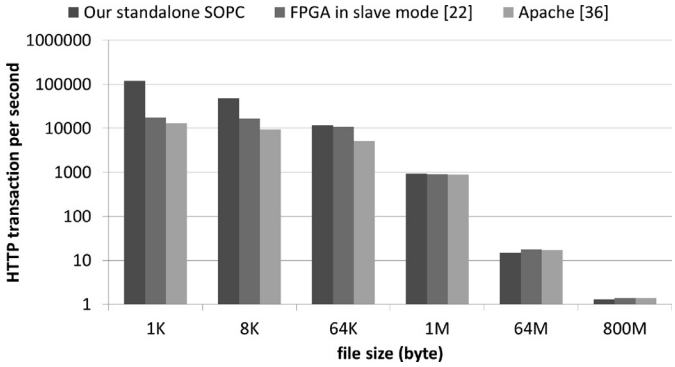


Fig. 12. HTTP transaction rate at different file sizes.

the bottleneck. For master-slave based implementation, a lot of message interaction and remote memory access operations via PCI-E were needed between the master side and the slave side, which put more delay during single HTTP transaction. For Apache based implementation, OS operations such as system call, user space to kernal space memory copy and interrupt handling bring more delay. In our standalone SOPC based web server, however, latency is reduced due to the tightly-coupled architecture. Compare with [21], our TCP/IP stack also has a similar throughput (8.3Gbps) as [21] (8.6 Gbps).

6.3.2. Number of HTTP transactions per second

Fig. 12 illustrates our explanation from another perspective. The HTTP transaction rate of the SOPC based web server can reach up to 120,000 TPS (Transactions per Second) when transmitting the same 1KB file, which is 6.8 times that of the master-slave based implementation and 9.2 times that of the Apache web server. However, as file size increases, the performance gap is getting narrower, and the throughput limitation is becoming the bottleneck.

We can use the performance model introduced from Section 2 to analyze the results. In master-slave based implementation system, the communication latency between the master side and the slave side via PCI-E interface is nearly 4μs. And during the process of a complete HTTP request-response operation including three-way handshake in the process of establishing a connection and four-way handshake in the process of disconnecting a connection, there are 12 communications operations over PCI-E interface. According to Eq. (6), the time consumed by control I/O operation T_{cio} is approximately equal to $4\ \mu s \times 12 = 48\ \mu s$. On the other hand, according to Eq. (7), the time consumed by data I/O operation T_{dio} is almost equal to $\frac{\Phi}{\theta_{PCI-E}} + \frac{\Phi}{\theta_{HardDisk}} \times cache\_miss\_rate$. When file size is small and cached in the memory of the master side, the T_{dio} is nearly equal to the PCI-E latency. The HTTP transaction rate of master-slave

based implementation is 17,500 TPS when file size is 1K byte, which means it needs 1000000 $\mu s/17500 = 57$ μs for each complete transaction. As per the computation, we can infer that most of the time is consumed by control I/O operations. This observation indicates that the master-slave based architecture is quite unsuitable for such situation. In our SOPC based implementation, however, the control I/O latency is small enough to be ignored, so that it dramatically reduces the process time for HTTP transaction, which is only 1000000 $\mu s/120000 = 8.3$ μs. This result indicates that our SOPC based web server can offer better user experience in terms of lower response time when suffering high concurrent requests.

6.3.3. Max number of concurrent connections

We continuously generated keep-alive HTTP requests to measure the max number of concurrent connections of three systems. TCP connections established based on such request do not tear down after requested file is completely transmitted. The number of max concurrent connections were recorded when the web server is not able to set up more TCP connections. Experimental results show that the max concurrent TCP connections supported by our stand-alone SOPC based web server, Apache web server and the master-slave based web server are nearly 100K, 80K, and 128 respectively. Since the QuickTCP [22] IP core only supports one TCP/IP session per instantiated pipeline, considering on-chip resource constrain, the max number of concurrent connections is limited for not able to integrate more pipelines. Also compared with [21], our TCP/IP stack supports much more concurrent TCP connections (100K VS. 10K), with similar performance and resource consumption.

6.3.4. Energy efficiency under peak performance

We use the term million bits transmitted per joule (Mb/J) to define the energy efficiency of different systems. The power consumption for one instance of the SOPC based web server running under maximum workload was 40W (including the power of the off-chip devices and the cooling fan). So the peak energy efficiency was calculated as $8320Mbps \div 40W = 208Mb/J$. However, the Apache web server consumed 160W with full workload and its peak energy efficiency was 58Mb/J, while the master-slave based implementation consumed 180W which leads to a worse peak energy efficiency that was 52Mb/J. In master-slave based web server, most of the energy was consumed by the conventional PC on the master side. Since the application mapped to such architecture does not have a large serial computation-intensive portion to be executed in the high performance general purpose processor, web servers based on the master-slave style architecture are obviously inefficient.

6.3.5. Energy efficiency in mimicry cluster

We measure the energy efficiency of our SOPC based web server in Mimicry Cluster with online dynamic scheduling. We emulated the dynamic changing requests where the peak workload is nearly 4 times of its average workload. 40 files ranging from 1MB to 64MB were used as requested files and 12 IBM x3635 M3 servers were used as client machines. To balance the QoS and dynamic energy efficiency, we empirically set the scheduling upper bound L_u to 85%, window size to 15 and scheduling interval to 5s. The experiment was carried out for 24 hours, and at the point of request peak, 12 FPGA were used to provide web services. A cluster that contains 12 master-slave based web servers and a cluster that contains 12 Apache web servers were tested under the same input. On account of unsustainable cost that frequently powers up and shuts down the conventional PC, we made all 12 nodes awake during the test.

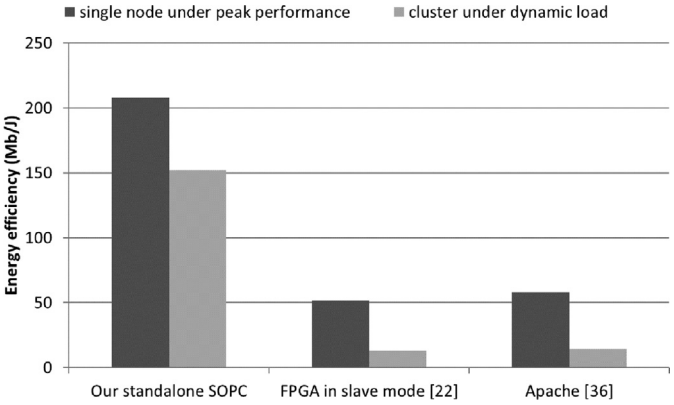


Fig. 13. Comparison of energy efficiency among our standalone SOPC implementation, FPGA-in-salve-mode implementation and Apache implementation of webserver. We evaluate the energy efficiency of both single node under peak performance and cluster under dynamic load.

Table 2
Latency of the user defined application logic in cloud based ECG classification application.

Data acquisition	Feature extraction	Neural network
4096	9728	61

We measured the total energy consumption $P(J)$ and total file sizes $S(MB)$ downloaded during the test period, then we calculated the energy efficiency as $8S/P(Mb/J)$. Fig. 13 shows the energy efficiency of the three systems under dynamic changing workload. The energy efficiency of our standalone SOPC based web server cluster was 152Mb/J, which was 73.8% of its peak energy efficiency(energy efficiency under maximum workload). However, the energy efficiency of the master-slave based web server was 12.8Mb/J, which was 24.7% of its peak energy efficiency. Moreover, Apache cluster achieved 26.4% of its peak energy efficiency. The results indicate that, using online dynamic scheduling, our SOPC based system is able to effectively utilize the processing power of activated computing node, while power off the rest of nodes to save energy. Thus, it gains much more energy efficiency in a dynamic way, compared to other two static scheduled systems.

6.4. Performance evaluation of the cloud based ECG classification in FPGA

In our study of cloud based ECG classification, 23 ECG records were selected from the MTI-BIH arrhythmia database [37] which is widely used in research community of ECG analysis. All records were digitized at 360 samples per second with 11-bit resolution for slightly over 30 minutes. These records include six ECG beat types: the normal beat (N), the left bundle branch block beat (LBBB), the right bundle branch block beat (RBBB), the atrial premature beat (APB), the premature ventricular contraction (PVC), and the paced beat (PB). For each beat type, we choose 50% beats for training and another 50% beats for testing. The parameters related to the neuron network were trained offline using software and then fixed to the implementation in FPGA. Ten FE pipelines were integrated in feature extraction module to maximize the throughput and make full use of reconfigurable resources. The size of the block ram in each FE pipeline is 16KB which can store 8K points of the ECG data in 16bit wide. Half of the memory space is used for current processing, while the other half is used for new ECG data buffering.

Table 2 illustrates the latency of the user defined application logic in our design. The latency of the FE pipeline to process a 4K points ECG data containing 12 heart beats is 9782 cycles. Al-

though the latency of reading data from the DDR3 memory to the block is 4096 cycles, such data acquisition latency was hidden in the execution pipeline using double buffering. Since the system clock is 156.25Mhz, the throughput of single FE pipeline is $8KB \div (9728cycles \times 1/156.25Mhz) = 128.5MB/s$. Considering the max theoretical upstream throughput of 10Gb Ethernet, 10 FE pipelines are sufficient to meet the requirement of the system level throughput: $(10Gb/s \div (128.5 \times 8)Mb/s = 9.72 < 10)$. On the other hand, the latency of the neural network module to classify one heart beat is 61 cycles. As a result, its throughput can be calculated as $8KB \div (61cycles \times 1/156.25Mhz \times 12beats) = 1.71GB/s > 10Gb/s$, which also meet the requirement of the system level throughput.

To simulate the behavior of users in client side, we used computers to generate multiple TCP connections to the server side, and continually upload ECG data that from the testing set. Each connection uploaded 4K sequential sample points to the server side. Classification of every heart beat was sent back to the users. Since the method and the experiment data we used for ECG classification referred to the work presented in [32], here we do not evaluate the classification accuracy which was already described in that paper. However, we focused on the performance related to the system throughput in a cloud based environment. Since the performance results of the method were not reported in that paper, in order to draw a comparison, we implemented a software version of the same application on IBM x3635 M3 server with Intel x5650 CPU (2.66 Ghz), 8GB DDR3 memory and Intel X520-SR2 10GE network adapter. The software version was programmed in standard C using Linux socket for network communication.

6.4.1. Maximum transactions per second

We used maximum transactions per second (Tps) as the performance metric to evaluate the system level throughput. A completed transaction included the uploading of the 4K points ECG data, ECG data classification and the sending back of the result. We gradually increased the speed of input requests generated from the computers until the system under test failed to respond, then we recorded the maximum speed of transactions. Our FPGA based solution was able to accomplish 41,830 transactions per second which is 38 times that of the software version. Compared to the software version, where both I/O-intensive workload and computation-intensive workload share the same CPU time, our standalone SOPC based architecture deals with two parts of workload by separated hardware modules, thus achieves a better performance.

6.4.2. Energy efficiency under peak performance

We use the term transactions per joule ($Trans/J$) to define the engergy efficiency of different systems. The power consumption for one instance of FPGA under maximum workload was $46W$. So the peak energy efficiency was calculated as $41830Tps \div 46W = 909Trans/J$. However, the software version consumed $172W$ under maximum workload and its peak energy efficiency was only $6.4Trans/J$.

6.4.3. Energy efficiency in mimicry cluster

We measure the energy efficiency of our cloud based ECG classification in Mimicry Cluster with online dynamic scheduling. We adopted the same experimental design used in that of the web server application. We measured the total energy consumption $P(J)$ and total completed transactions T during the test, then we calculated the energy efficiency as $T/P(Trans/J)$. As shown in Fig. 14, the energy efficiency of our standalone SOPC based cluster was $711Trans/J$, which was 78.2% of its peak energy efficiency. However, the energy efficiency of the software based cluster was $1.7Trans/J$, which was 27.2% of its peak energy efficiency. Thus, our standalone SOPC based clusters achieved $418X$ improvement in energy efficiency compared to commercial clusters.

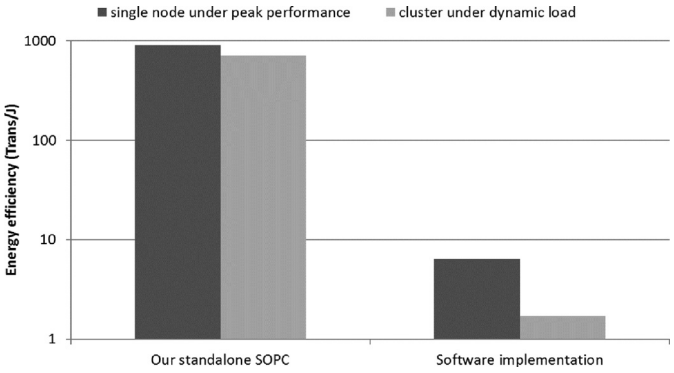


Fig. 14. Comparison of energy efficiency between our standalone SOPC implementation and software implementation of cloud based ECG classification. We evaluate the energy efficiency of both single node under peak performance and cluster under dynamic load.

Table 3
Benchmarking with the state-of-the-arts (ECG samples processed per second (Sams/s)).

Hashim[41]	Ieong[42]	Zhou[43]	Ours
29.06×10^3	57.53×10^6	17.75×10^6	63.89×10^6

We calculate the energy cost of two systems according to [38]. Each user generates 360 ECG samples per second, and each transaction consists 4000 samples. Since the unit price of electricity is 1 CNY per KWh in China, if the cloud system serves 1 million users (one seventh of cardiac patients in China), the annual energy cost is nearly $10^6 \cdot 12months \cdot 30days \cdot 24h \cdot 60min \cdot 60s \cdot (360/4000)Trans/s \cdot 1/(1.7Trans/J) \cdot (1 \cdot 2.78 \cdot 10^{-7})CNY/J = 457776CNY$ when using commercial servers, while $10^6 \cdot 12months \cdot 30days \cdot 24h \cdot 60min \cdot 60s \cdot (360/4000)Trans/s \cdot 1/(711Trans/J) \cdot (1 \cdot 2.78 \cdot 10^{-7})CNY/J = 1094CNY$ when using our SOPC based servers. The calculation shows that our SOPC based cloud computing system can significantly reduce the energy cost compared to traditional commercial servers.

6.4.4. Benchmarking with the state-of-the-arts

We compare the performance of our work in single node with prior FPGA solutions. The performance comparison results in terms of processing throughput (ECG samples processed per second (Sams/s)) is given in Table 3. Since most of the ECG processing systems [39–41] are designed for portable and single user scenario, they typically employ embedded processors or light-weight hardwire to save power and chip size. So that these systems have a moderate ECG processing capacity (e.g. Hashim [41] achieves only 29.06×10^3 Sams/s). Ieong [42] otherwise designed a throughput optimized QRS detection hardware in FPGA, which achieves 57.53×10^6 Sams/s. However, it only accelerated QRS detection while our work presents a whole cloud based ECG analysis system including network handling, connection management, QRS detection, feature extraction and classification. Zhou [43] proposed an FPGA-assisted cloud framework for massive ECG processing. It uses a commercial server to handle network connections and attaches an FPGA via PCIE to accelerate ECG processing. However, the data transmission latency between the PC host and the FPGA degrades the whole performance (17.75×10^6 Sams/s). Also, it does not contain feature extraction and classification logic. Compared with it, our system is a stand-alone SOPC system. By offloading both networking operations and ECG data analysis in a close-coupled architecture, our system can achieve 63.89×10^6 Sams/s.

6.5. Limitations

Although our system has an advantage in terms of performance and energy efficiency, it indeed has limitations. Since a complex cloud application may contain many functions that are not suitable or not necessary to be implemented in hardware, it is better to implement such functions in embedded microprocessor. However, the use of Microblaze limits the performance of dealing with such functions, thus becoming the bottleneck of the system. Fortunately, FPGAs of the latest generation such as kintex-7 have already addressed this problem by integrating a hard core of dual ARM processor which provides more powerful computing capability (unavailable to purchase when this project began). Our proposed architecture can be perfectly ported to these new devices, which we believe are more suitable for cloud computing in terms of usability. The second limitation is that FPGA is harder to use compared to CPU. In IaaS (Infrastructure-as-a-Service) and PaaS(Platform-as-a-Service) cloud scenarios, developing any application using high-level synthesis or Verilog/VHDL is a difficult process for general users. However, architectures proposed in our paper are mainly focus on SaaS (Software-as-a-Service) cloud scenarios, where cloud providers undertake the work of architecture design and implementation in FPGA while general users only receive cloud services without awareness of the use of FPGAs in the cloud. In this situation, the cost down of energy consumption in the cloud is much more valuable than the price paid to use FPGA. Still, our work will be a good starting point to accelerate the development of SOPC based cloud applications with high performance network access capability, mass storage and cloud service functionalities.

7. Conclusion

In this paper we presented the design and implementation of a standalone SOPC based architecture that tightly couples the network IO handling and data processing in a single FPGA to accelerate cloud applications. A massive-sessions optimized TCP/IP hardware stack that supports 10Gb Ethernet and 100K concurrent TCP connections is proposed to meet the requirement of high concurrent requests. Due to the hardware pipeline implementation of network protocols and user defined application logics, our implementation can achieve higher performance, lower energy consumption with standalone cloud service functionalities. Moreover, we proposed an online dynamic scheduling method in standalone SOPC based reconfigurable cluster system, where single FPGA can be scheduled on or off at a relatively low cost according to the realtime workload variance, thus saved overall energy in the dynamic process. Two case studies including a web server cluster and a cloud based ECG classification cluster were implemented on our newly built reconfigurable prototype platform, which is a parallel system that explored the performance potential of FPGA as standalone systems. Experimental results showed that the webserver implemented in our system is able to support more concurrent TCP connections with lower response delay and similar throughput compared with two other state of the art webservers. In cloud based ECG classification application, 38X speed up in performance and 418X speed up in energy efficiency were achieved compared to the software based cloud systems. These measured results are consistent with the analytical results of our performance analysis.

Acknowledgments

This paper is partially sponsored by National High-Technology Research and Development Program of China (863 Program) (No. 2009AA012201 and No. 2015AA050204), National Natural Science Foundation of China (61373032), and National Research Foundation (NRF), Prime Ministers Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. Prof. Qiu is partially supported by NSF CNS 1359557.

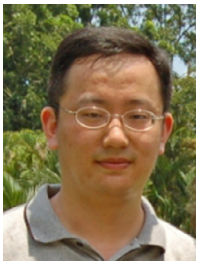
References

[1] M. Friedewald, O. Raabe, Ubiquitous computing: an overview of technology impacts, Telematics Informatics 28 (2) (2011) 55–65.
[2] X. Wu, X. Zhu, G.-Q. Wu, W. Ding, Data mining with big data, IEEE Trans. Knowl. Data Eng 26 (1) (2014) 97–107.
[3] K. Lim, D. Meisner, A.G. Saidi, P. Ranganathan, T.F. Wenisch, Thin servers with smart pipes: designing soc accelerators for memcached, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ACM, 2013, pp. 36–47.
[4] L. Ganesh, H. Weatherspoon, T. Marian, K. Birman, Integrated approach to data center power management, IEEE Trans. Comput. 62 (6) (2013) 1086–1096.
[5] S.-H. Hung, T.-T. Tzeng, J.-D. Wu, M.-Y. Tsai, Y.-C. Lu, J.-P. Shieh, C.-H. Tu, W.-J. Ho, Mobilefbp: designing portable reconfigurable applications for heterogeneous systems, J. Syst. Archit. 60 (1) (2014) 40–51.
[6] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, A. Foong, Tcp onloading for data center servers, IEEE Comput, 37 (11) (2004) 48–58.
[7] J. Yan, Z.-X. Zhao, N.-Y. Xu, X. Jin, L.-T. Zhang, F.-H. Hsu, Efficient query processing for web search engine with fpgas, in: Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2012, pp. 97–100.
[8] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, et al., A reconfigurable fabric for accelerating large-scale datacenter services, in: Proceedings of the 41st Annual International Symposium on Computer Architecture, IEEE, 2014, pp. 13–24.
[9] Y. Gao, H. Guan, Z. Qi, B. Wang, L. Liu, Quality of service aware power management for virtualized data centers, J. Syst. Archit. 59 (4) (2013) 245–259.
[10] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, R.P. Doyle, Managing energy and server resources in hosting centers, in: Proceedings of the 18th ACM symposium on Operating systems principles, 35, ACM, 2001, pp. 103–116.
[11] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, F. Zhao, Energy-aware server provisioning and load dispatching for connection-intensive internet services., in: Proceedings of the 5th USENIX symposium on Networked Systems Design and Implementation, 8, USENIX, 2008, pp. 337–350.
[12] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM Trans. Auton. Adapt. Syst. 3 (1) (2008) 1–39.
[13] M. Hosseinabady, J.L. Nunez-Yanez, Energy optimization of fpga-based stream-oriented computing with power gating, in: Proceedings of the 25th International Conference on Field Programmable Logic and Applications, IEEE, 2015, pp. 1–6.
[14] C.-S. Li, H. Franke, C. Parris, B. Abali, M. Kesavan, V. Chang, Composable architecture for rack scale big data computing, Future Gener. Comput. Syst. 67 (2017) 180–193.
[15] J. Baliga, R. Ayre, K. Hinton, R. Tucker, Green cloud computing: balancing energy in processing, storage, and transport, Proc. IEEE 99 (1) (2011) 149–167.
[16] S.R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, M. Margala, An fpga memcached appliance, in: Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM, 2013, pp. 245–254.
[17] M. Blott, K. Vissers, Dataflow architectures for 10gbps line-rate key-value-stores, in: Proceedings of the 25th IEEE Hot Chips Symposium, IEEE, 2013, pp. 1–25.
[18] Z.-Z. Wu, H.-C. Chen, Design and implementation of TCP/IP offload engine system over Gigabit Ethernet, in: Proceedings of the 15th International Conference on Computer Communications and Networks, IEEE, 2006, pp. 245–250.
[19] H. Jang, S.-H. Chung, D.K. Kim, Y.-S. Lee, An efficient architecture for a tcp offload engine based on hardware/software co-design., J. Inf. Sci. Eng. 27 (2) (2011) 493–509.
[20] T. Uchida, Hardware-based tcp processor for gigabit ethernet, IEEE Trans. Nucl. Sci. 55 (3) (2008) 1631–1637.
[21] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, R. Carley, Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware, in: Proceedings of the 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2015, pp. 36–43.
[22] PLDA Inc, QuickTCP IP core for Xilinx reference manual, 2015. URL http://www.plda.com/products/fpga-ip/xilinx/fpga-ip-tcpip/quicktcp-xilinx.
[23] Dini Group, TCP offload engine IP for latency critical, FPGA-based embedded networking applications, 2014. URL http://www.dinigroup.com/new/TOE.php.
[24] K. Benkrid, Y. Liu, A. Benkrid, A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 17 (4) (2009) 561–570.
[25] F. Belletti, M. Cotallo, A. Cruz, L.A. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, et al., Janus: an FPGA-based system for high-performance scientific computing, Comput. Sci. Eng. 11 (1) (2009) 48–58.
[26] X.-H. Sun, Y. Chen, Reevaluating amdahls law in the multicore era, J. Parallel Distrib. Comput. 70 (2) (2010) 183–188.
[27] L. Cherkasova, P. Phaal, Session-based admission control: a mechanism for peak load management of commercial web sites, IEEE Trans. Comput. 51 (6) (2002) 669–685.

[28] A. Rosenthal, P. Mork, M.H. Li, J. Stanford, D. Koester, P. Reynolds, Cloud computing: a new business paradigm for biomedical information sharing, J. Biomed. Inform. 43 (2) (2010) 342–353.
[29] J. Ekanayake, T. Gunarathne, J. Qiu, Cloud technologies for bioinformatics applications, IEEE Trans. Parallel Distrib. Syst. 22 (6) (2011) 998–1011.
[30] İ. Güler, et al., Ecg beat classifier designed by combined neural network model, Pattern Recognit. 38 (2) (2005) 199–208.
[31] S. Kadambe, R. Murray, G.F. Boudreaux-Bartels, Wavelet transform-based qrs complex detector, Trans. Biomed. Eng. 46 (7) (1999) 838–848.
[32] S.-N. Yu, Y.-H. Chen, Electrocardiogram beat classification based on wavelet transformation and probabilistic neural network, Pattern Recognit. Lett. 28 (10) (2007) 1142–1150.
[33] S. Himavathi, D. Anitha, A. Muthuramalingam, Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization, IEEE Trans. Neural Netw. 18 (3) (2007) 880–888.
[34] J. Wu, Meaning and vision of minic computing and minic security defense, Telecommunications Science (in Chinese) 7 (2014) 2–7.
[35] SPIRENT, Avalanche 3100 datasheet, 2015. URL http://www.spirent.com/Test-solutions_datasheets/Broadband/PAB/Avalanche/Avalanche_3100_datasheet.
[36] Apache, Apache HTTP Server Version 2.2 Documentation, 2015. URL http://httpd.apache.org/docs/2.2/en/.
[37] G.B. Moody, R.G. Mark, The impact of the mit-bih arrhythmia database, Eng. Med. Biol. Mag. 20 (3) (2001) 45–50.
[38] V. Chang, G. Wills, A model to compare cloud and non-cloud storage of big data, Future Gener. Comput. Syst. 57 (2016) 56–76.
[39] Y. Zou, J. Han, S. Xuan, S. Huang, X. Weng, D. Fang, X. Zeng, An energy-efficient design for ECG recording and r-peak detection based on wavelet transform, IEEE Trans. Circuits Syst. II Express Briefs 62 (2) (2015) 119–123.
[40] C.-I. Ieong, P.-I. Mak, C.-P. Lam, C. Dong, M.-I. Vai, P.-U. Mak, S.-H. Pun, F. Wan, R. Martins, A $0.83-\mu w$ QRS detection processor using quadratic spline wavelet transform for wireless ECG acquisition in $0.35-\mu m$ cmos, IEEE Trans. Biomed. Circuits Syst. 6 (6) (2012) 586–595.
[41] A. Hashim, C.Y. Ooi, R. Bakhteri, Y.W. Hau, Comparative study of electrocardiogram qrs complex detection algorithm on field programmable gate array platform, in: Proceedings of the 2014 IEEE Conference on Biomedical Engineering and Sciences, IEEE, 2014, pp. 241–246.
[42] C. In Ieong, M.I. Vai, P. Un Mak, Ecg qrs complex detection with programmable hardware, in: Proceedings of the 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, IEEE, 2008, pp. 2920–2923.
[43] S. Zhou, Y. Zhu, C. Wang, X. Gu, J. Yin, J. Jiang, G. Rong, An fpga-assisted cloud framework for massive ecg signal processing, in: Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing, IEEE, 2014, pp. 208–213.

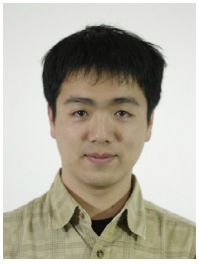
Xu Wang received the B.Eng. degree in electronics science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2005. He is currently pursuing the Ph.D. degree in computer science and technology from Shanghai Jiao Tong University, Shanghai, China. His current research interests include reconfigurable computing, computer architecture, machine learning and big data.

Yongxin Zhu is an Associate Professor with the School of Microelectronics, Shanghai Jiao Tong University, China. He is a senior member of China Computer Federation and a senior member of IEEE. He received his B.Eng. in EE from Hefei University of Technology, and M.Eng. in CS from Shanghai Jiao Tong University in 1991 and 1994 respectively. He received his Ph.D. in CS from National University of Singapore in 2001. His research interest is in computer architectures, embedded systems, medical electronics and multimedia. He has authored and co-authored over 90 English journal and conference papers and 30 Chinese journal papers. He has 18 Chinese patents approved.



Yajun Ha received the B.S. degree in electrical engineering from Zhejiang University, Hangzhou, China, in 1996, the M.Eng. degree in electrical engineering from the National University of Singapore (NUS), Singapore, in 1999, and the Ph.D. degree in electrical engineering from Katholieke Universiteit Leuven, Leuven, Belgium in 2004. Between 1999 and 2004, he did his Ph.D. research project at IMEC, Leuven. He has been an Assistant Professor in the Department of Electrical and Computer Engineering, NUS, since 2004. His research interests lie in the embedded system architecture and design methodologies, particularly in the area of reconfigurable computing. He holds one U.S. patent and has published more than 50 internationally refereed technical papers in his areas of interest.

Meikang Qiu received the B.E. and M.E. degrees from Shanghai Jiao Tong University, China. He received the M.S. and Ph.D. degrees in Computer Science from University of Texas at Dallas in 2003 and 2007, respectively. He had worked at Chinese Helicopter R&D Institute and IBM. Currently, he is an assistant professor of CS at Pace University. He is an IEEE Senior member and has published 140 journal and conference papers, including 15 IEEE/ACM Transactions. He is the recipient of the ACM Transactions on Design Automation of Electronic Systems (TODAES) 2011 Best Paper Award. He also received three other best paper awards (IEEE EUC'09, IEEE/ACM GreenCom'10, and IEEE CSE'10) and one best paper nomination. He also holds 2 patents and has published 3 books. He has also been awarded SFFP Air Force summer faculty in 2009. He has been on various chairs and TPC members for many international conferences. He served as the Program Chair of IEEE EmbeddCom'09 and EM-Com'09. His research interests include embedded systems, computer security, and wireless sensor networks.



Tian Huang is a Research Associate with Cavendish Laboratory, University of Cambridge, United Kingdom. He received his B.Eng in EE and Ph.D. in EECS from Shanghai Jiao Tong University in 2008 and 2016 respectively. His research interest is in High Performance Computing, Signal Processing, Embedded System, and Artificial Intelligence.

Xueming Si is a professor and deputy director of Shanghai Key Laboratory of Data Science, Fudan University, China. His research interests include computational mathematics, high performance computing and computer security. He has authored over 20 journal and conference papers. He filed 7 Chinese patents among which 1 have been approved.

Jiangxing Wu is a professor of the PLA Information Engineering University. He is currently an Academician of the Chinese Academy of Engineering. His research interest includes Communication Network and Switching Technology. So far, he has completed more than ten national key projects in the field of communication technology in China. The representative of the engineering achievements includes: the first program-controlled switches in China; the first advanced intelligent network system in China; high performance broadband information network (3Tnet). He has won two First Prizes of the National Sci-Tech Advance Award in China.