Service-Oriented Architecture on FPGA-Based MPSoC

Chao Wang, Xi Li, *Member, IEEE*, Yunji Chen, Youhui Zhang, *Member, IEEE*, Oliver Diessel, and Xuehai Zhou

Abstract—The integration of software services-oriented architecture (SOA) and hardware multiprocessor system-on-chip (MPSoC) has been pursued for several years. However, designing and implementing a service-oriented system for diverse applications on a single chip has posed significant challenges due to the heterogeneous architectures, programming interfaces, and software tool chains. To solve the problem, this paper proposes SoSoC, a service-oriented system-on-chip framework that integrates both embedded processors and software defined hardware accelerators s as computing services on a single chip. Modeling and realizing the SOA design principles, SoSoC provides well-defined programming interfaces for programmers to utilize diverse computing resources efficiently. Furthermore, SoSoC can provide task level parallelization and significant speedup to MPSoC chip design paradigms by providing out-of-order execution scheme with hardware accelerators. To evaluate the performance of SoSoC, we implemented a hardware prototype on Xilinx Virtex5 FPGA board with EEMBC benchmarks. Experimental results demonstrate that the service componentization over original version is less than 3 percent, while the speedup for typical software Benchmarks is up to 372x. To show the portability of SoSoC, we implement the convolutional neural network as a case study on both Xilinx Zynq and Altera DE5 FPGA boards. Results show the SoSoC outperforms state-of-the-art literature with great flexibility.

Index Terms—Service-oriented architecture, multiprocessor, system on chip

1 Introduction

Multi-Core has been a mainstream microprocessor implementation technique, especially for high-performance computing. In data-intensive application fields, it is now becoming increasingly popular to use Field Programmable Gate Arrays to accelerate the state-of-the-art applications, such as genome sequencing, data mining, and deep learning algorithms [1]. As more processors and heterogeneous Intellectual Property (IP) accelerators are being integrated into a single chip to build Multi-Processor Systems on Chip (MPSoC) platforms, the computational capability is increasingly powerful, which makes it possible to provide highly efficient platforms for diverse applications [2]. For example, the Intel QuickAssist Technology Accelerator Abstraction Layer introduces a software framework for deploying platform-level services and abstracting the interconnect technology from the application code. This software abstraction layer allows the accelerators to be transparently shared amongst multiple workload clients.

However, cutting-edge MPSoC design methodologies aim to improve the raw performance of embedded systems, while disregarding the flexibility and portability across different target architectures. Consequently, most MPSoC researchers suffer from inconvenient programming models, high design complexity, and low productivity when they design middleware and prototype chips for diverse applications. Since instruction-set architectures (ISA), programming interfaces and tool-chains of different processors are significantly different from each other [3], how to improve the flexibility and portability remains an extremely challenging problem.

To tackle this problem, we propose a method for introducing service-oriented architecture (SOA) concepts to the MPSoC design paradigm [4], [5]. Traditional SOA provides good flexibility and extensibility at low cost by providing reusable modules. Moreover, SOA can largely reduce the complexity of integration and application development by providing well-defined package interfaces [6]. With all these benefits, the SOA concept has been widely applied in software services, web services, and even operating systems design [7]. It is naturally capable of combining different processing elements (PEs) through the well-defined interfaces and without concerning the programmer with the implementation of hardware platforms, operating systems, and programming languages. Therefore, the SOA-based design is an efficient way of quickly constructing prototyping systems.

As a consequence, we claim that adopting SOA concepts into MPSoC platforms has two significant advantages. First, SOA architecture can easily integrate numerous computing

- *C. Wang, X. Li, and X.Zhou are with the University of Science and Technology of China, Hefei 230027, Anhui, China. E-mail: {cswang, llxx, xhzhou}@ustc.edu.cn.*
- *Y. Chen is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: cyj@ict.ac.cn.*
- *Y. Zhang is with the Department of Computer Science, Tsinghua University, Beijing 100084, China. E-mail: zyh02@tsinghua.edu.cn.*
- *O. Diessel is with the University of New South Wales, Sydney NSW 2052, Australia. E-mail: odiessel@cse.unsw.edu.au.*

Manuscript received 23 Oct. 2015; revised 26 June 2016; accepted 25 Sept. 2016. Date of publication 5 May 2017; date of current version 13 Sept. 2017. (Corresponding authors: Xi Li and Xuehai Zhou).
Recommended for acceptance by Y. Lu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2701828

1045-9219 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

TABLE 1
Brief Summary for SOA and MPSoC Related Research Areas

Type	Related work & References	Benefits	Drawbacks
SOA	Mobile computing system [8]	1) Modularity 2) Flexibility	1) Inadequate performance 2) Does not directly apply to MPSoC and chip design
	Enterprise architecture [9]	3) Scalability	3) No dynamic reconfiguration
	Electronic productions [10]	4) Programmability	
	Workflow Composition [11] Operating Systems [7]		
MPSoC	ReconOS [12], Hthreads [13], RecoBus [14], [15], FlexCore [16], OneChip [17], ReMAP [18], RAMP [19], MOLEN [20], Accelerator [21]	1) Modest performance with heterogeneous architecture 2) Flexible IP core integration 3) Reconfigurable feature	1) Does not readily support high-level programming 2) Does not automatically support service substitution
SOA+MPSoC	SOMP [4]	Advantages of both SOA and MPSoC concepts	

resources together; therefore it facilitates building heterogeneous research platforms. By using this feature, MPSoC can benefit from the strengths of each PE type so as to provide high-performance computing capability for diverse applications. Second, since the structural programming interfaces in SOA architecture are well defined, SOA can provide a unified API even when the hardware is reconfigured. This feature can not only help researchers conveniently add/remove computing elements, but also accelerates the process of constructing system prototypes and evaluating overheads.

While there are many state-of-the-art research projects related to SOA and MPSoC platforms individually, few studies have so far been conducted into fusing SOA and MPSoC concepts. In this paper, we propose SoSoC, which introduces the SOA model to system-on-chip design paradigms. Benefiting from the high computing performance of MPSoC and the flexibility of SOA, SoSoC can provide various services with structured application programming interfaces, based on embedded processors and hardware IP cores as fundamental hardware resources. Applications are divided into subtasks and are scheduled to embedded processor or IP blocks at run-time. To evaluate the performance of SoSoC, we build a prototype system on FPGA development board. *We claim the following contributions:*

1) A service-oriented model for heterogeneous MPSoC: this paper proposes a novel hierarchical SOA model consisting of multiple layers suitable for MPSoC. SOA concepts provide structured programming and well-defined service integration interfaces, thereby facilitating the construction of MPSoC prototypes for diverse applications.
2) Adaptive mapping with dynamic reconfiguration: this paper presents an adaptive service mapping and out-of-order scheduling method based on an MPSoC hardware architecture. The integrated PEs of the MPSoC can be reconfigured so as to adapt to applications. When hardware reconfiguration is ready, tasks can be automatically remapped and spawned to IP cores for parallel execution.
3) Prototype implementations and experiments: To evaluate SoSoC, we implemented an MPSoC prototype on a state-of-the-art Xilinx FPGA development board using Microblaze processors and heterogeneous IP accelerators. Experimental results demonstrate the service componentization overheads of SoSoC are less than 3 percent, and the peak speedup achieves 370x for EEMBC Benchmarks.

The remainder of the paper is organized as follows. Section 2 summaries the related approaches. Section 3 discusses the detailed architecture and methodology of SoSoC, including architecture, hierarchical model, scheduling, interconnect, and programming interfaces. The FPGA prototype implementation is outlined in Section 4. Section 5 presents the experimental results and their analysis. Section 6 illustrates a case study using convolutional neural networks on both Xilinx and Altera FPGA platforms. Finally, we conclude the paper and pinpoint some future directions in Section 7.

2 Background

It is common knowledge that SOA has been successfully exploited in high-level software models, while MPSoC is generally utilized as multiprocessing hardware and architecture platforms. However, few studies are focusing on integrating SOA and MPSoC together. Nevertheless, there are lots of related works of each area which motivate our research. Table 1 lists the main related works.

First of all, various SOA frameworks have been developed for software engineering, web services, operating systems, such as mobile computing system [8], enterprise architectures [9], electronic productions [10] and scientific workflow composition frameworks [11]. From these approaches, we can summarize the major advantage of SOA is to encapsulate different computing resources and package them into

a unified service access interface. Thus these service-based approaches provide better flexibility and extensibility with lower cost through reusable software modules. In particular, in [8], Thanh and Jørstad provide a presentation of SOA for mobile services. Haki and Forte [9] demonstrate that using the SOA concept in an enterprise architecture (EA) framework makes the best of the synergy existing between these two approaches. Delamar and Lastra [10] present an array of architecture patterns for creating distributed message frameworks, focusing mainly on globally distributed federations and locally distributed clusters. Meanwhile, attention has shifted towards lower level architectures, such as to operating systems [7] and multiprocessor platforms [4]. Similar to SOA-based approaches, SWAP [22] is a component-based parallelization framework that uses specification compatibility graphs to abstract and model algorithms between high-level specifications and low-level implementations.

In contrast to SOA, the original design goal of MPSoC is provide an implementation platform for application-specific designs, particularly for embedded systems. With the rapid development of semiconductor technologies and devices like FPGAs, MPSoC is now able to integrate sufficient computing resources to build a supercomputing engine on a single chip. With appropriate configuration and optimization, MPSoC can achieve very high-performance levels. Previous research has focussed on hardware-software partitioning [23], scheduling [24], interconnection [25], and communication mechanisms [26]. While these studies have made specific contributions on certain aspects, the problem of how to design a flexible and efficient platform and a prototype system is still worth pursuing.

Experimental MPSoC platforms have been verified on heterogeneous computing platform, such as OneChip [17], ReMAP [18], RAMP [19], MOLEN [20] and Accelerator [21]. These studies focus on reconfigurable and heterogeneous computing-paradigms including maximizing raw performance along with softer evaluation metrics such as flexibility, programmability, and power utilization. However, each platform is constructed using specific hardware resources and a specific toolchain, which makes it rather difficult to port the applications from one to another. Moreover, most of these studies aim at application-specific hardware design, which means programmers need to acquire detailed knowledge of the system specification and implementation to be able to handle the tasks mapping, scheduling, and distribution manually. The degree of automatic parallelization is therefore still worth investigating.

Along with the prototype platforms targeting specific hardware, there are some well-known reconfigurable hardware infrastructures: ReconOS [12], for instance, demonstrates hardware/software multithreading methodology on a host OS running on the PowerPC core of modern FPGA platforms. Hthreads [13], RecoBus [14] and [15] are also state-of-the-art FPGA-based reconfigurable platforms. Besides FPGA-based research platforms, FlexCore [16] is an alternative approach based on a general-purpose multicore platform that is similar to the one used in our SoSoC study.

In contrast to the application constraints of the bookkeeping techniques of FlexCore, SoSoC proposed in this paper, is a general-purpose framework supporting a wide range of task acceleration engines. In particular, to enhance the scalability and modularity beyond simply incorporating a diversity of IP accelerators, this work introduces SOA concepts into reconfigurable MPSoC design. Since SOA can provide flexibility and extensibility for MPSoC chip design at lower cost in the design process, thus SoSoC can decrease the MPSoC design complexity across a wide range of hardware accelerators with negligible overheads. Based on SoSoC, researchers could focus on further studies of scheduling algorithms, interconnection schemes, and reconfigurable technologies, etc. Furthermore, SoSoC can also reduce the burden of MPSoC architects and shorten the time to market of chips.

Before introducing the SoSoC architecture, we first define the following terms.

Tasks. Throughout this paper, we use the term *tasks* to refer to pure functional instances such as an IDCT and AES running on specific hardware IP modules. Note that the granularity of a task as defined in this paper is different from general task definitions with threads. When SoSoC processes a task, it will be treated as a specific service. Control information (e.g. task ID, target servant, etc.) as well as the requisite operands are transferred through first-in-first-out (FIFO) based hardware links between the scheduler and servants.

Services. Services are defined as different functionalities that are accessible to users. All services are packaged in a function library and invoked by standard function calls. All the services are launched and provided by servants.

Servants. Servants refer to functional modules dedicating to provide one or several services. Servants are classified into different categories as follows:

Application Servants. Application servants are responsible for providing application programming interface (API) and the run-time environment. Moreover, application servants are also in charge of task profiling to locate the hotspots of applications. The profiling information can facilitate the dynamic re-mapping and re-scheduling of a task.

Scheduling Servants. Scheduling servants are employed in task partitioning, mapping, and run-time scheduling. Regarded as the kernel component, a scheduling servant plays a key role in the online exploration for task level parallelism. It receives the task sequence from application servants and then detects inter-task data dependencies. Whenever input parameters and hardware are available, the task can be immediately issued.

Computing Servants. Computing servants are designed to run computing services and can be further classified into hardware or software computing servants. On the one hand, each software computing servant runs on a microprocessor with dynamic software function libraries. On the other hand, a hardware servant is implemented at register transfer level (RTL) and then packaged as an IP core that can only do a very specific kind of service.

3 Architecture and Concepts

3.1 Introducing SOA to MPSoC architecture

By introducing SOA into MPSoC architecture design, the traditional primitives are abstracted as below: 1) Each task can be regarded as an extended special instruction. By that means, the original application consisting of multiple tasks can be abstracted as an instruction sequence. 2) Each

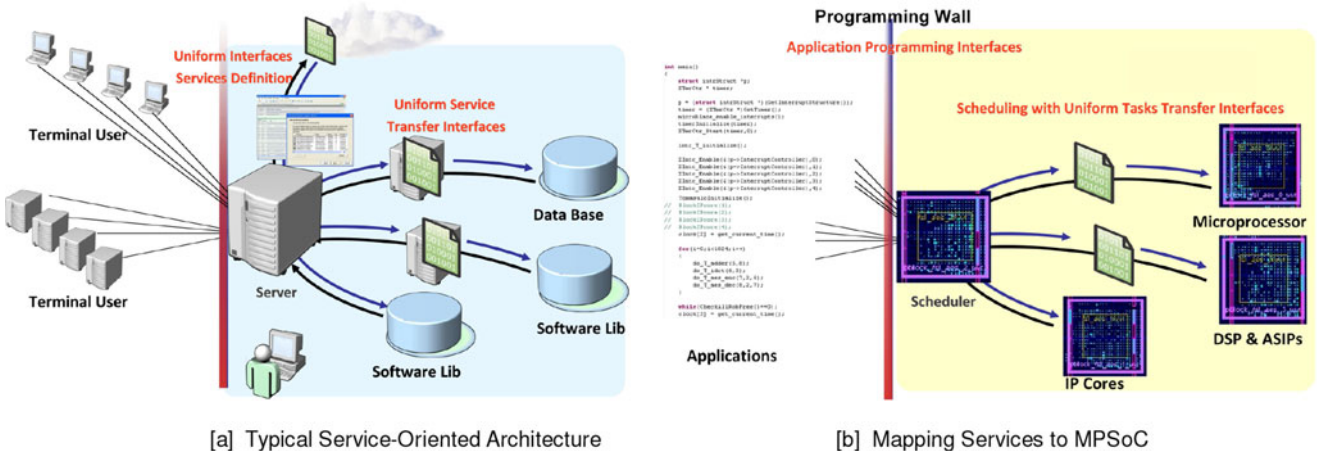


Fig. 1. Typical SOA concepts and the corresponding services in MPSoC.

processor or IP core can be regarded as a dedicated functional unit to run an abstract instruction. Each abstract instruction is scheduled to a certain functional unit by the SoSoC middleware, either in static or dynamic ways.

Fig. 1a illustrates the typical framework of traditional SOA concepts. The front-end terminal users access the services via a uniform interface with service definitions, whereby each service is packaged and exposed in an API-like manner. Meanwhile, in the back-end, each service is composed of specific functionalities provided by software libraries and databases through uniform service interfaces. The functionality for each service is composed or combined from multiple data resources with a service scheduling mechanism. It should be noted that the service composition and scheduling are invisible to front-end terminal programmers.

The proposed SOA mapping onto an MPSoC hardware platform is illustrated in Fig. 1b, where the service definition interfaces are realized as APIs, and microprocessors, DSPs and hardware IP core function as service providers. The application is first decomposed into multiple services, which are then scheduled at runtime. Whenever a pending service has obtained its requisite input parameters, it can be offloaded to a certain PE for immediate execution.

3.2 SoSoC Architecture and Components

The SoSoC architecture is illustrated in Fig. 2. SoSoC is based on a hardware platform that can provide heterogeneous multi-core resources such as processors, DSPs, FPGAs and

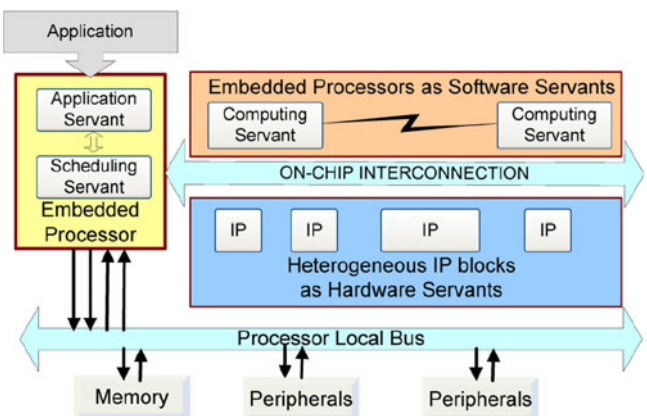


Fig. 2. SoSoC architecture is based on MPSoC architectures.

others. In particular, SoSoC is composed of the following components: an application servant, a kernel scheduling servant, and several embedded processors as software computing servants, intellectual property (IP) cores as hardware servants, interconnect modules, buses, memory blocks, and various peripheral modules. In particular, the responsibilities for each type of modules are as follows:

1. An application servant runs on a general-purpose processor to provide the basic run-time environment and APIs to tasks. Moreover, it also profiles and traces the application runtime information and sends all service requests to a scheduling servant for further processing.
2. A scheduling servant is in charge of task partitioning, mapping, distribution, scheduling and task transmission.
3. Software servants: each task should be distributed to either a software or hardware servant at run-time. Of the two types of manifestations, all the software services are provided by general-purpose processors with function libraries. In general, this type of servant can run different kinds of tasks. Every software servant has access to a homogeneous service library which contains the available services for the system. The scheduler can dispatch the tasks to different computing servants considering the current workload of the system.
4. Hardware servants: in contrast to the software servants, each hardware servant accelerates only one specific kind of task. SoSoC can integrate a variety of heterogeneous hardware IP or ASIP cores at a time, depending on the available hardware resources on the chip. Also, IP cores can be dynamically reconfigured according to application demands.
5. Interconnect modules: on-chip interconnect utilized for data transfer between the scheduling servant and computing servants. The data includes service control requests and input/output results. When the underlying platform is FPGA-like, a variety of interconnect topologies can be implemented.
6. Memory and peripherals, such as I/O, debugging interface, UART controller, timer controller and interrupt controller, are connected to the scheduler

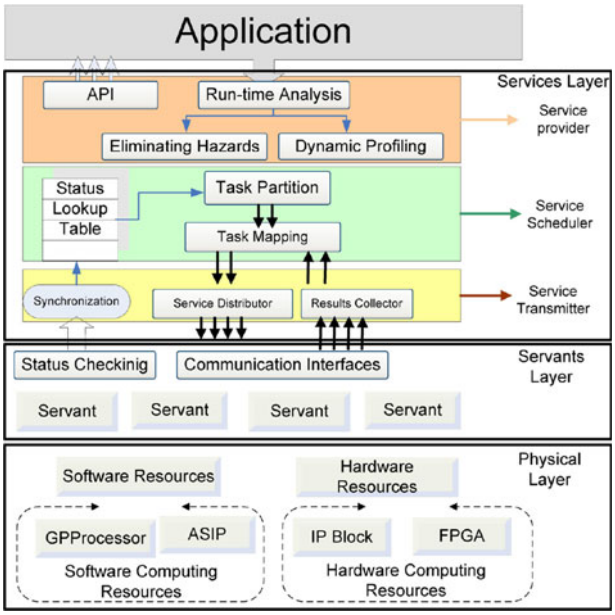


Fig. 3. Hierarchical level model of SoSoC is comprising three layers.

via a bus. These peripheral devices can realize a complete system and aid programmers in operating a debug interface.

3.3 Hierarchical Model

Each computing servant is either implemented as software to be executed on a GPP or as hardware on accelerators via IP cores. Based on the SoSoC hardware, we construct an SOA hierarchical model, as illustrated in Fig. 3. This model consists of three layers: services layer, servants layer and physical layer, which will be detailed respectively.

(1) Services Layer. The services layer is composed of three modules: services provider, scheduler and transmitter.

First of all, the services provider exposes application programming interfaces and run-time application analysis services to programmers. The API is invoked by users for sending task requests during execution and returns the status of the currently running task as feedback. The run-time application analysis includes trace and profiling. A trace module is used for keeping track of the services requests. Meanwhile, a dynamic profiler can be activated to locate and store the hotspots of the program. The information of hotspots can guide the reconfiguration of different IP engines for performance acceleration. What's more, to improve the task level parallelism, inter-task hazards are detected and eliminated.

Second, the service scheduler is in charge of allocating the application to services and mapping each service to a target computing servant. First, during execution, one application is divided into several sub-tasks, each of which is abstracted to a specific service and is dynamically mapped and scheduled either to a software or hardware servant, according to the system requirements and computing servants' status. The status of all computing servants is recorded in a status lookup table. We use a task queue for hardware and software services. When a task is mapped, it first looks up the queue and calculates the expected execution time for execution in either hardware or software: 1) the time waiting to be scheduled as software, 2) the communication overheads between the scheduling processor and hardware computing services. The algorithm can thereby make a wiser choice before the task is dispatched to a certain service.

Finally, a service transmitter dispatches the service request to heterogeneous computing servants including embedded processors and IP hardware accelerators. The service distribution has a consistent interface irrespective of the service type or target servant. When the service is finished, results are also collected by the service transmitter. Furthermore, to maintain the runtime status, a synchronization module has been integrated to obtain the traces for the hardware platform.

(2) Servants Layer. As described above, services are dynamically mapped to different computing servants for parallel execution. All the servants are managed for efficient use and load balancing. The data transmitted between the servants layer and the services layer include services requests, input/output parameters, and execution results. A status checking interface is provided to the service layer for synchronization.

Computing servants include hardware and software computing servants. Because hardware servants can obtain higher performance than software servants in most cases, so the tasks are always scheduled to hardware if there are free hardware IP cores.

(3) Physical Layer. Finally, all the servants are implemented in software resources executed by processors or hardware resources executed by IP accelerators. On one hand, a software servant mainly consists of two parts: a general purpose processor core or ASIP (ARM, PowerPC, MicroBlaze, etc.) and a software library loaded into the processor. Consequently, every computing servant is capable of supplying different kinds of services that SoSoC provides to programmers. On the other hand, hardware servants are implemented as IP cores, coarse-grained reconfigurable arrays (CGRA), or reconfigurable logic units (RLU). Each IP core or RLU can be reconfigured for specific applications.

3.4 Hardware Tasks-to-Servants Arbitration

Task partitioning and scheduling methods play a vital role in architectural supports. Before tasks are offloaded to IP cores, OoO middleware should identify the target processor to run the current task, and also decide when the task can be issued.

3.4.1 Task to Servants Mapping

In this paper, static core modules and reconfiguration modules (RMs) are implemented separately, of which only RMs are reconfigured at run-time to reduce the bitstream downloading overheads. In task partition and scheduling layer, reconfiguration libraries are integrated. After IP cores are reconfigured, tasks mapping and scheduling strategies need to be reconsidered. Therefore a task-to-core table is employed to identify the target IP core, as described in Fig. 4. The table maintains a mapping of tasks to cores to virtualize the selection of the destination core. Each table entry contains the task ID currently running on that core as well as a count of the number of issued tasks destined for that core. When a new IP core is deployed, the table elements will be flushed and updated.

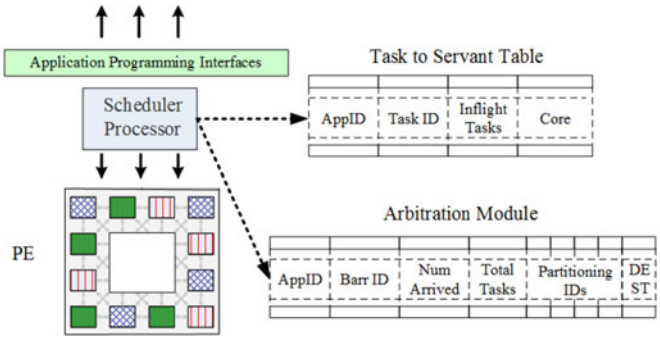


Fig. 4. Task to Servant Table and Arbitration Module.

When a task is issued, it obtains the core currently assigned as its destination core in the table; and it stores its results to the appropriate output queue upon completion. A side effect of this table based approach is that instructions will not issue to the fabric if the destination core is not available. This prevents the producing task from filling up the fabric if the consumer is not present. Even with the table, however, spawned tasks could accumulate in the fabric if the current task forces are switched out while data is in flight to it, which would require the consumer to be switched back into the same core to receive the values. To prevent this situation, the task-to-core mapping table maintains a counter for the number of in-flight tasks destined for each core. On a request to switch out, the scheduler checks the number of in-flight tasks bound to its core. If this is greater than zero, the fabric is blocked from accepting any new tasks destined for that core and the core continues to execute until the in-flight counter reaches zero. At this point, the application can be stalled and the fabric unblocked.

For each IP core, the specific task execution time, speedup, area cost and power consumption information are also maintained by the scheduler. The information will assist scheduler to make task partition decisions and to achieve better load-balancing status and higher throughputs. Since FPGA is an area-constrained platform, different IP cores are competing for the limited hardware resources. For task scheduling, tasks are also considered to be arranged in sequences, which should improve the throughput as well as FPGA area efficiency.

3.4.2 Barrier Synchronization

Barriers are one of the most common synchronization operations. However, with a typical memory-based implementation, the overhead of executing a barrier can be significant, especially as the number of cores increases. This overhead prevents the use of barriers at fine granularities. In cases where a barrier is followed by a serial function that is performed by one of the tasks and the output communicated to all participating tasks, the scheduler may directly synthesize the function into the fabric with the output communicated to the participants' output parameters.

To implement barriers for synchronization, a barrier table is integrated to ensure that all the returning tasks must not be allowed to issue to the fabric until all participating cores have arrived at the barrier, as presented in Fig. 4. To achieve this, each core participating in the barrier loads some value(s) into its input queue. Once the loads from all of the cores have reached the head of their respective input queues and all tasks, have indicated arrival at the barrier. The Barrier Table also determines that all tasks have arrived at the barrier, with information related to each active barrier. Each table contains as many entries as cores attached to a PE cluster, which includes both general processors (denoted in the central white block), and heterogeneous accelerators (described in colored blocks). The table keeps track of the total number of tasks, the number of arrived tasks, and the cores that are participating in the barrier. The number of arrived tasks and participating cores are updated whenever a task arrives, meanwhile the total and arrived task counts are compared to determine when to issue a task. In a system with multiple PE clusters, a dedicated bus communicates barrier updates among clusters. The bus transmits the barrier ID as well as the associated application ID. All tasks participating in a barrier must be actively running for all input data to be available. Each table entry maintains a list of the IDs of the local tasks that are participating in the barrier as well as a bit indicating if they are actively running. If a barrier is ready to be released but not all participating tasks are active, the scheduler controller triggers an exception to switch the missing tasks back in. Once all tasks are available, the barrier can proceed

3.5 Service Out-of-Order Scheduling

The compiler ensures that the hardware scheduler only deals with task sequences without control dependencies. Based on the programming model described in the previous section, an out-of-order task scheduler is implemented in the middleware layer to uncover task-level parallelism. For demonstration, we have implemented an MP-Tomasulo algorithm [27], [28], which dynamically detects and eliminates inter-task write after write (WAW) and write after read (WAR) dependencies, and thus speeds up the execution of the whole program. With the help of our MP-Tomasulo algorithm, programmers need not take care of the data dependencies between tasks as these are automatically eliminated for them. Listing 1 shows the formal description of the MP-Tomasulo algorithm, which is divided into four stages as follows:

Issue: The head task of Task Issuing Queue is in *Issue* stage if an RS-table entry and an ROB entry are both available. If yes, they may be stored in an ROB entry (Lines 5-6) or the VS-table (Line 9). In these cases, just copy the variables to the allocated RS-table entry. Otherwise the input variables may not be available due to RAW dependency; in this case, the task records which task will produce the needed variables (Line 7). For all output variables, VS-table is updated indicating that the newest value of the output variables will be produced by the issuing task (Lines 11-15). Besides, the information of the allocated ROB entry and the RS-table entry will be updated (Lines 13, 16-17).

Execute: If all the input variables of a task are prepared, then the task can be distributed to the associated PE for execution immediately. Otherwise, the RS-table builds an implicit data dependency graph indicating which task will produce needed variable. Once all the input variables of a task are ready, the task is spawned to the corresponding PE. If there is more than one task that satisfies the *Execute* stage requirement, an FCFS strategy is applied.

Write Results: When a PE completes a task, it sends results back to the Task Receiving Queue of MP-Tomasulo module, updates ROB and RS-table. For each task in RS-table, if the input variables are produced by the completed task, the associated RS-table entry is updated with the results (Lines 5-6).

Listing 1. MP-Tomasulo Algorithm in Pseudocode.

MP-Tomasulo Algorithm

Issue Stage
requirement:
RS-Table(r) and ROB(b) both available
Action:
```
1. for(i = 1; i< = num_rd_set; i++){
2. index = rd_set[i];
3. if(VS[index].Busy){
4.    h = VS[index].Recorder, t = VS[index].ROBorder;
5.    if(ROB[h].Ready){
6.       RS[r]. V[i] = ROB[h]. Value[t];RS[r].Q[i] = 0;}
7.    else {RS[r].Q[i] = h;}
8.    }
9. else {RS[r].V[i] = index; Data; RS[r].Q[i] = 0;}
10. }
11. for(j = 1; j< = num_wr_set; j++){
12.    index = wr_set[j];
13.    ROB[b].Dest[j] = index; VS[index].Rcorder = b;
14.    VS[index].Busy = yes; VS[index].ROBorder = j;
15. }
16. RS[r].Busy = yes; RS[r].Dest = b;
17. ROB[b].Busy = yes; ROB[b].Ready = no;
```

Excute Stage
requirement:
for all i in[1,num_rd_set] {RS[r].Q[i] = = 0}
Action:
Distribute the task to associated PE and compute results

Write Results Stage
requirement:
execution done at r
Action:
```
1. b = RS[r].Dest; RS[r].Busy = no;
2. for(j-1; j< = num_wr_ser; j++){
3.    ROB[b]. Value[j] = results[j]; b = RS[r].Dest[j];
4.    for(i = 1; i< = num_rd_set; i++){
5.    ∀ x(if(RS[x].Q[i] = = b)){
6.       RS[x]. V[i] = results[j]; RS[x].Q[i] = 0;}
7. }
8. ROB[b].Ready = yes;
```

Commit Stage
requirement:
task at the head of the ROB(entry h) and
ROB[h].Ready = = yes
Action:
```
1. for(j = 1; j<num_wr_set; j++){
2.    d = ROB[h]. Dest[j];
3.    VS[d]. Data = ROB[h]. Value[j];
4.    if(VS[d]. Rcorder = = h) {VS[d]. Busy = no;}
5. }
6. Rob[h]. Busy = no;
```

Commit: Update VS-Table and write results to disks in the order as tasks are issued. The tasks stored in ROB are in the same order as they are issued, so the consistency of data stored on disks is kept.

To add or remove IP cores conveniently, we introduce a software/hardware co-design methodology. All IP cores are packaged within a structural interface based on the requirements of the physical on-chip interconnect. Whenever the architecture changes, the interfaces in the header file need to be changed accordingly, but the user applications do not need to be modified or recompiled.

3.6 Programming Interfaces

SoSoC provides two types of programming interfaces: both blocking and non-blocking. The principles of each kind of programming interface are described in Listing 2.

Listing 2. An Example of Annotated Codes in the Programming Model.

```
/*– # SoSoCLib.h –SoSoC Lib Description – */
#pragma input(idct in) output(idct_out)
void do_T_idct(int idct_out[N], idct_in[N]);
#pragma input (aes_in1,aes_in2) output(aes_out)
void do_T_aes(int aes_out[M], aes_in1[M], aes_in2[M]);
/*–#Main Program on Scheduler Processor– */
#include "SoSoCLib.h"
main (){
. . .. . .
do_T_idct(idct_out, idct_in);
do_T_aes (aes_out, aes_in1, aes_in2);
. . .. . .
}
```

Listing 2 Outlines an example of annotated codes in the programming model. Generally, there are two parts inside the example:

1) The top part of Listing 2 gives an example of a SoSoC library that provides dedicated internal service functions. The annotation indicates the do_T_idct and do_T_aes functions can be executed on IP cores, with the directionality described for each operand.
2) The bottom part of Listing 2 illustrates an example of the main program running on a scheduling processor. The main application code is identical to a sequential implementation using library functions. What's required by the programmer is only to include the SoSoC library as header files. The programming model maps the annotated functions to the target processor or IP core. The codes in automatically parallelized regions work as normal codes without annotations using the functions already defined in the included library.

At runtime, whenever the user application reaches a call site to one of the internal functions, the main program packs all the task operand values and transfers the data to the middleware layer for mapping and scheduling. As the execution of the main program is decoupled from the execution of the tasks, it can resume execution to continue spawning the following tasks. The middleware layer, on the other hand, asynchronously detects the task dependencies, and schedules tasks when they are ready.

TABLE 2
Services and Related Applications in EEMBC

Services	App	Description	EEMBC
Adder	Adder	Data Accumulation	AutoBench
IDCT	IDCT	Inverse DCT	AutoBench
RGB2YUV	JPEG	Color Space Converter	ConsumerBench
2DIDCT	JPEG	2D Inverse DCT	ConsumerBench
Quant	JPEG	Quantization	ConsumerBench
AES ENC	AES	AES encryption module	DENBench
AES DEC	AES	AES decryption	DENBench
DES ENC	DES	DES encryption	DENBench
DES DEC	DES	DES decryption	DEN Bench

As the task executes, a run-time profiling mechanism is integrated into the system to locate the hot spots of each application. The hot spot information helps the programmers locate which parts are running with most frequently and for how long. The hotspot information can be used to guide the selection of task accelerators for performance optimization.

4 Prototype System Implementation

4.1 Platform Setup

To measure the performance and overheads of SoSoC, we implemented a prototype system on a Xilinx XUPV5 board equipped with a Virtex-5 XC5VLX110T FPGA. We utilized MicroBlaze (MB) version 7.20.A (with a clock frequency of 125 MHz, the local memory of 8 KB, no configurable instruction or data cache) as our general purpose processor. The whole environment was built and set up using Xilinx ISE Design Suite. The SoSoC prototype was implemented on a single FPGA. Two MB processors were utilized; one was employed as the scheduling servant, and the other was used as a computing servant. In total, we implemented 9 hardware IP cores as hardware servants illustrates a demonstration system with 4 computing servants). Each computing servants was connected with the scheduling servant through a pair of FSL links. Each MB had its instruction and data cache implemented in BlockRAM. We used a processor local bus (PLB) to connect peripherals, including an interrupt controller, a UART controller, and a timer controller. We implemented the SoSoC with the following components:

(1) A scheduling servant is implemented on an MB processor. The scheduling algorithm and mapping schemes were implemented in a software kernel.
(2) Software computing servants were also constructed on individual MB processors. Service functions were encapsulated in standard C libraries. APIs in the C language were provided to users.
(3) Hardware computing servants were implemented in function blocks implemented in HDL and packaged as standalone IP cores.
(4) The scheduling servant was connected to the software computing MB and IP cores via FSL links. Task requests and results were transferred via FSL buses.

Synplify Pro and Xilinx ISE were employed to estimate the area utilization and power consumption for the FPGA fabric. To compute a rough estimate of the area, we adopted a metric of CLB tile area from the model by Kuon and Rose [29]. The model reports that the area of a CLB tile with 10 6-input LUTs in the 65nm technology node is approximately 8,069 μm^2. We used this estimate of 807 μm^2 per LUT and multiplied it by the total number of LUTs occupied by our design to generate an area estimate. Furthermore, we utilized the associated XPower Analyzer of the Xilinx FPGA toolchain to estimate the power consumption.

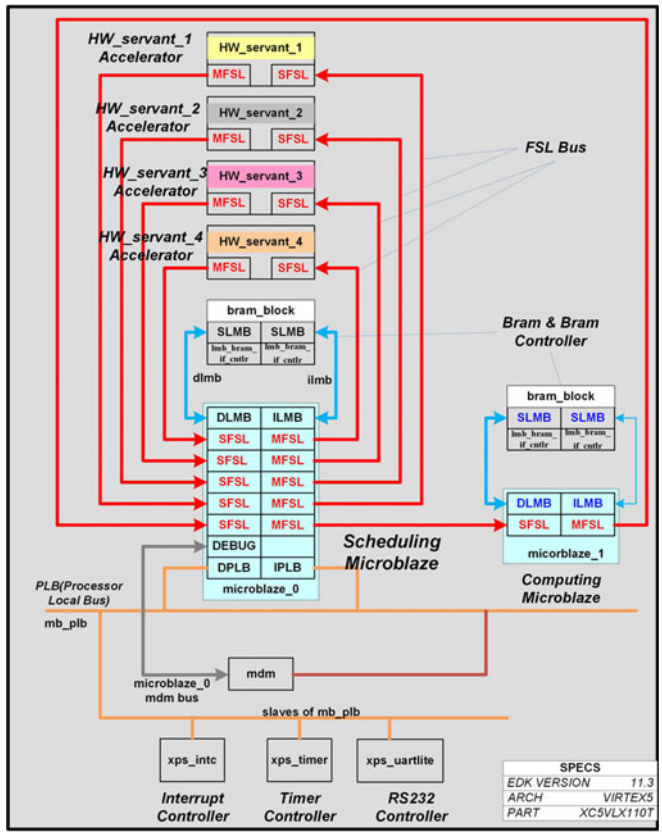


Fig. 5. SoSoC Prototype in FPGA.

4.2 Integrated Services

After the general purpose processor is selected, we designed 9 services from EEMBC, as shown in Table 2, to measure the functionality and performance. For each service, software and hardware servants were both implemented. The high-level block diagram of a typical case study of AES hardware servants is illustrated in Fig. 5. To support dynamic partial reconfiguration, different services are packaged in a similar manner and attached to the FIFO interfaces via the same group of FSL signals. Both input buffer and output buffer data structures are employed to store the I/O parameters locally as they are transferred one by one in the FIFO channels. As the data in a FIFO can only be read using a stream-like pattern, it was possible to implement a common control logic module the control logic module handled the standard service operations including: 1) read input from FIFO to input buffer, 2) execute service, 3) store results to output buffer, and 4) write results from output buffer to FIFO. This common structure provided a universal interface pattern to allow servants to be reconfigured at run-time. On the distinct functionalities, data accumulation is responsible for summing the input digits, IDCT module includes a multiply operation followed by an accumulation, while AES includes

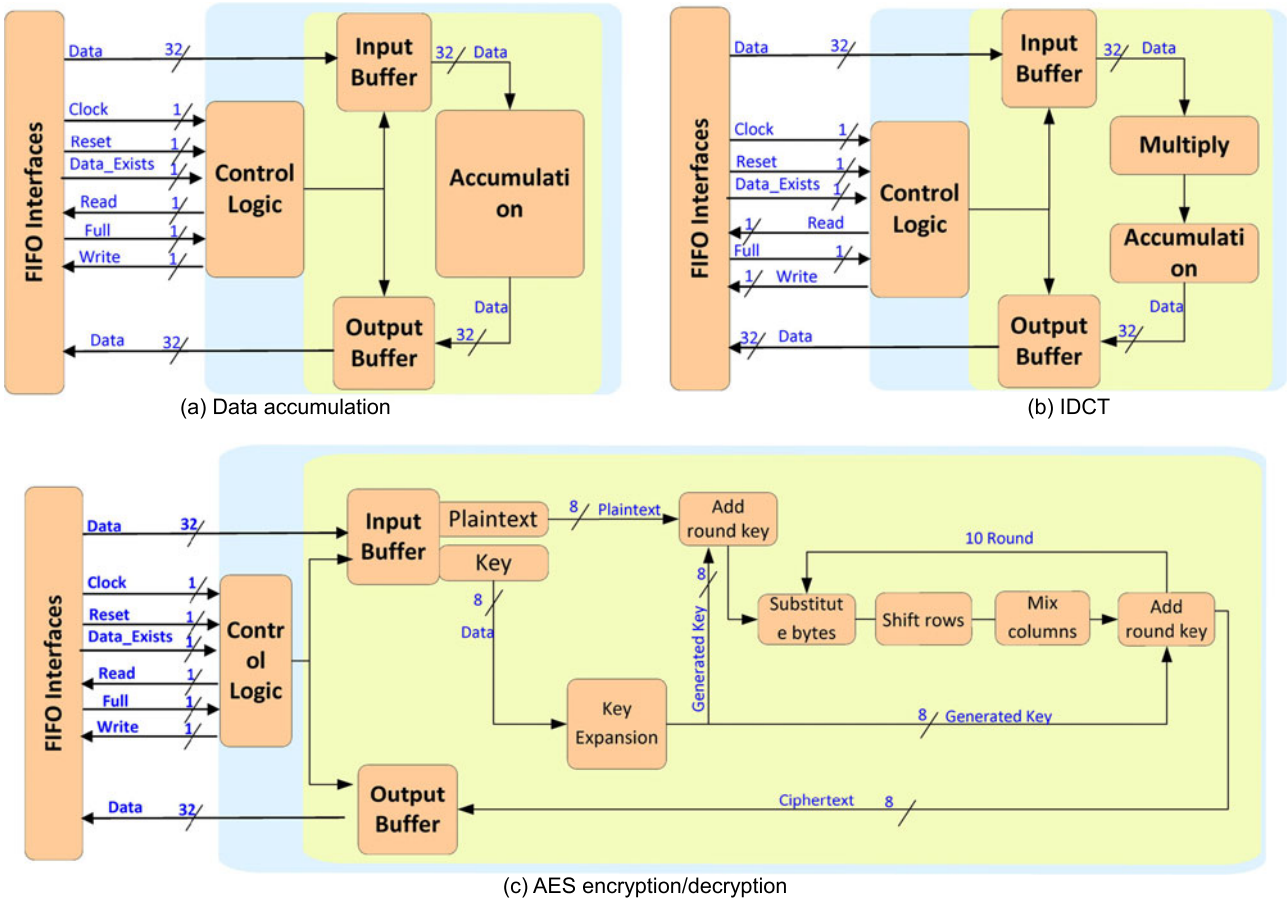


Fig. 6. Data accumulation, IDCT, and AES hardware servants.

one key expansion module and 10 rounds of encryption/decryption steps.

5 Experimental Results and Analysis

Based on the prototype, we used parts of the EEMBC benchmarks to measure the scheduling overheads of the SoSoC architecture. Different servants were integrated into the platform to measure the speedup under different circumstances. The number of processors and IP cores were reconfigured according to application needs. We used similar evaluation criteria to those of [22], which included componentization overheads, speedup and hardware costs.

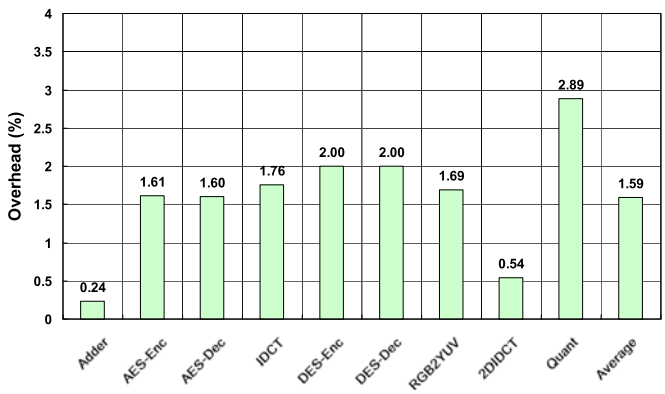


Fig. 7. Overhead of componentization over the original version.

5.1 Overhead of Service Componentization

We first investigated the performance overhead of service componentization. For this purpose, we compared the performance of the original (unmodified) sequential application on the componentized software services and the original uncomponentized software. We define the overhead as the percentage increase in execution time of the componentized version over the original version. Fig. 6 presents the measurements we took. The highest overhead was 2.89 percent for Quant, and all other programs exhibited only negligible overheads.

5.2 Speedup of Parallel Hardware Services

5.2.1 Speedup for Sequential Applications

To evaluate the speedup achieved by the SoSoC hardware services, we evaluated the hardware speedup over the software execution and the task sequences including batches of tasks.

Using SoSoC, we generated hardware versions corresponding to nine EEMBC software programs (Adder, AES_Enc, AES_Dec, IDCT, DES_Enc, DES_Dec, RGB2YUV, 2DIDCT, and Quant). The hardware speedups were computed on the original (unmodified) software version, and were found to range from 1.55x to 373x (see Fig. 7). Each hardware service was attached to an MB processor with a pair of FSL bus channels. As the AES and DES have large-scale computational complexity they achieved the highest speedup.

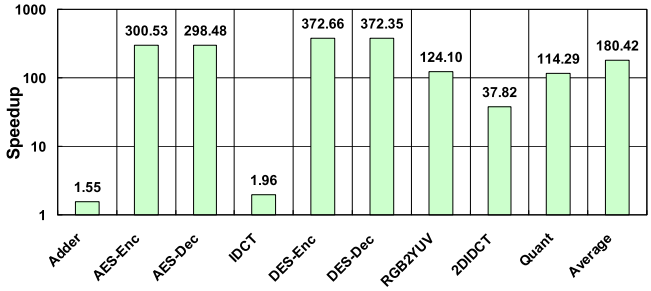


Fig. 8. Performance improvement due to service substitution.

5.2.2 Speedup for Parallel Applications against Sequences

To measure the maximum speedup for SoSoC, we integrated up to 4 identical computing servants simultaneously. We investigated the parallel execution mode and sequential mode, as plotted in Fig. 8.

For the parallel execution mode, the execution time on each servant was able to be completely overlapped; therefore the ideal speedup is 4.0x. For the sequential mode, data hazards such as read-after-write (RAW) could not be resolved by scheduling, which results in the speedup < 1.0x.

The X-axis refers to the total number of tasks. As the number of executed tasks grows, the speedups asymptotically reach their maximums. The maximum speedup for the 4-unit system is 3.74x, and the result for a single unit is 0.957x, which means that even with the scheduling and communication overheads, the experimental values can achieve 93.6 and 95.7 percent of the ideal peak values.

5.3 Scalability Analysis

In this Section, we analyze the scalability of the SoSoC platform, which includes two parts: 1) How SoSoC performed when the hardware service became increasingly powerful, and 2) How SoSoC performed when more hardware computational kernels were integrated.

5.3.1 Services at Different Speedup

To measure the influence of hardware IP cores with different efficiencies, we constructed a platform consisting of two modules: one scheduling MB with one IDCT hardware module. The speedup of the system was assessed as the relative hardware/software execution time was varied. The task scale 8∼256 indicates the total number of the IDCT tasks.

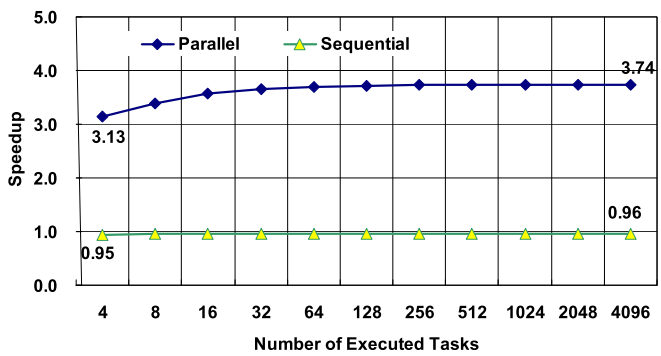


Fig. 9. Peak speedup against execution in a single core.

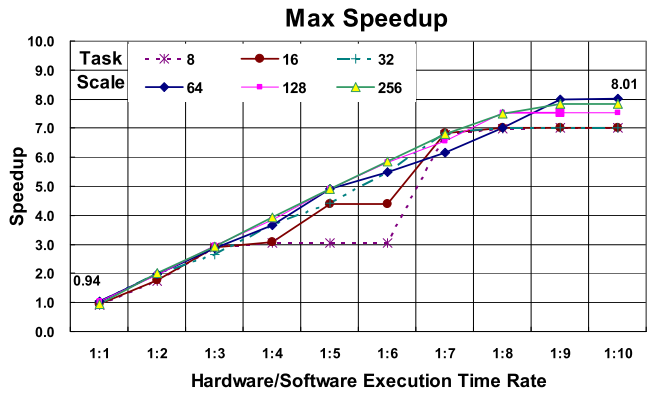


Fig. 10. The impact of Different Hardware Execution Time

Fig. 9 demonstrates the experimental results of different hybrid systems. As the relative execution time of hardware to software was increased from 1:1 to 1:10, the observed speedup also increased in a roughly linear relation. When the integrated hardware computing servant had the same efficiency as software (speedup of 1.0x), the speedup of the SoSoC system was found to be 0.94x, due to the scheduling overheads, while when the hardware was set to operate at 10.0x the software speed, the speedup of the SoSoC system was found to be 8.01x. The experimental results demonstrate that the SoSoC system is very stable and that the performance scales with hardware computational kernels of diverse performance.

5.3.2 Scalability Analysis with Number of Services

Given that the prototype was built on a reconfigurable FPGA platform, we were also able to measure the scalability with different numbers of hardware computing servants. We integrated 1 software computing servant and N hardware IP cores ($N = 0, 1, 2, 3, 4 \ldots$). We used real data from Fig. 10 to assess the scalability with all 9 hardware services. For each hardware service, we included the hardware services incrementally into the platform by reconfiguring the FPGA (with from one to four replica modules), and measured the speedup, respectively.

Fig. 11 reports the speedup with a different number of hardware IP cores. We considered the execution time on a single IP core as the baseline. When there was one Microblaze with one IP core, the speedup was found to be less than 1.0x due to the scheduling and communication overheads. When four identical hardware servants were integrated,

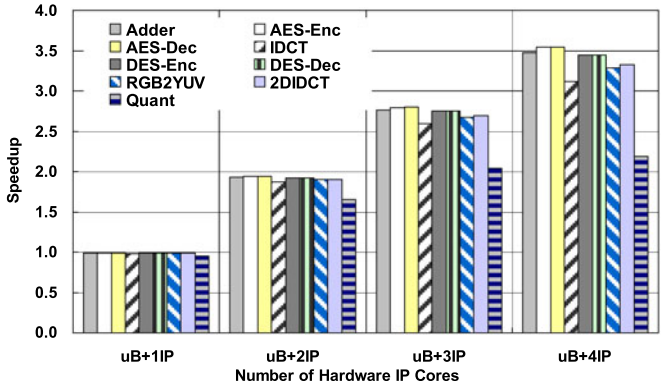


Fig. 11. Speedup with different number of hardware IP Cores.

TABLE 3
Sample Task Sequence

Task number	Task type	Source variables	Destination variable
T_1	DCT	a	b
T_2	AES_DEC	e	d
T_3	IDCT	b	f
T_4	AES_ENC	d, g	h
T_5	QUANT	b, c	g
T_6	DES ENC	g	i

the speedup increased to as much as 3.54x. The experimental results demonstrate that our SoSoC system could provide good scalability when more computational kernels are involved.

5.4 Discussion of the Out-of-Order Scheduling

In this section, we use a simple case study to illustrate the processing flow of our out-of-order scheduling method MP-Tomasulo. The test case is described in Table 3, and the experimental timing diagram is described in Fig. 12.

At the model start-up, the token representing the task T1 is generated and is then dispatched. The transition checks the state of computational kernels in the modeled system and assigns the DCT services to the task T1.

In the second time unit, task T2 is generated. As T2 has no task dependencies with T1, it can immediately be assigned to the AES_DES computational kernel.

In the third time unit, the task T3 is generated. At that time, the sources variable "b" is not ready due to T1 is not finished. Thus a read-after-write (RAW) data dependence is identified and the task T3 stalls.

As time goes on, the first task T1 will accomplish its execution and write the result back into its destination variable "b". Upon detecting the presence of the "b", the RAW data dependency is resolved. In Fig. 12, RAW data dependence is represented by the arrowed line which connects T1 and T3, while the stall period caused by it is represented by the bar tagged with Stall-1 in Fig. 12.

When T4 enters the Issue stage, the source variable "d" is found not ready yet. It indicates that there is dependency between the tasks T2 and T4. The latter task T4 will be stalled until T2 has returned the variable "d" during its Commit stage. Similar to T3, the stall period Stall-3 in T5 is caused by the RAW data dependencies. The task T5 can

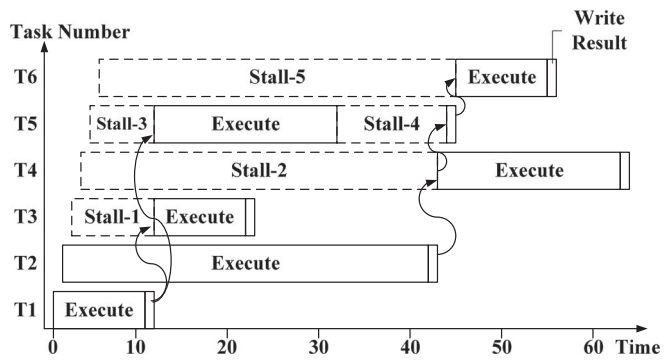


Fig. 12. Timing Diagram of the sample task sequence in Table 3.

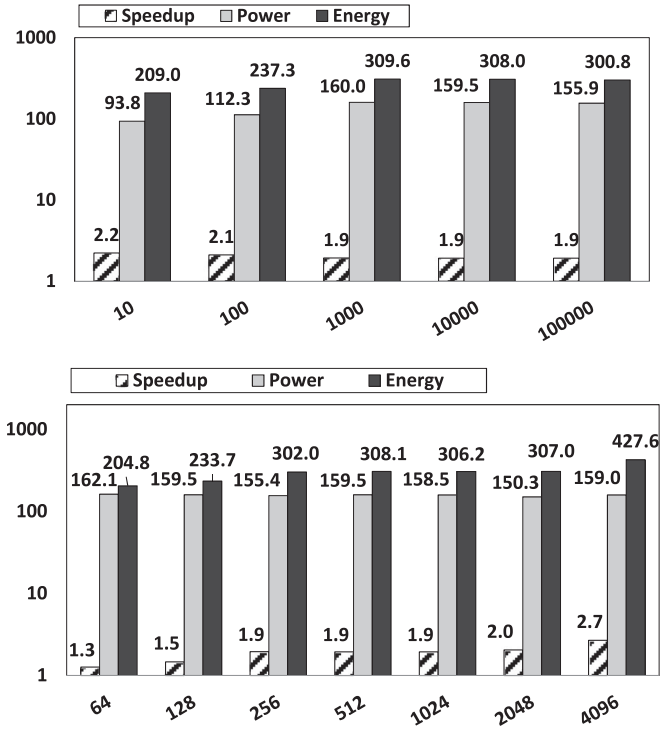


Fig. 13. Experimental results of hardware arbitration with different data sizes and task scales.

begin to execute in before the task T4 since it is not dependent on other tasks. When T5 accomplishes the execution and intends to write its result, the Write Result will check the absence of its destination variable "g". Since the task T4 has stalled and the variable "g" is not read by T4, the task T5 is stalled. It indicates the existence of anti-dependence. The stall period will last until the task T4 has fetched its input data. After the task T5 issues, T6 is allowed to enter the scheduler. However, T6 cannot be executed due to the RAW data dependence. As the result, the task T6 stalls. The stall period will last until the task T5 has written the result. After that, the task T6 will accomplish its execution.

By investigating the timing diagram, we can get an overview on how tasks interact with each other and maintain the data dependencies in our scheme. To this end, we confirm that our Out-of-order scheduling scheme can correctly schedule tasks with data dependencies to exploit parallelism.

5.5 Evaluation of Hardware Arbitration

To evaluate the performance of hardware arbitration, we measured the speedup, power, and energy for hardware arbitration and traditional software arbitration schemes. Furthermore, these metrics are evaluated with different data sizes and task scales.

Fig. 13 illustrates the speedup, power, and energy consumption with different data scales. First, the hardware arbitration can achieve about 2.1x speedup comparing to the software arbitration. The speedup remains flat when the data size increases from 10 to 100,000. In comparison, the power and energy consumption increase with the data size. For example, the hardware arbitration can take 93.8x less power and 209x energy at 10 data size, and increases to 155.9x and 300.8x at 100,000, respectively. Results show that the software arbitration consumes 160x more power and

TABLE 4
Hardware costs, area and power of SoSoC architecture

IP cores	Description	LUTs	Area(uM^2)	FFs	BRAMs	Power(mW)
Scheduler MB	Scheduler	1650	1331,550	1489	0	9.7
Adder	Data aggregation	182	146,874	82	0	2.0
IDCT	Inverse Discrete Cosine Transform	215	173,505	85	0	14.8
AES ENC	AES encryption	1413	1140,291	790	3	17.8
AES DEC	AES decryption	1587	1280,709	788	3	16.8
DES ENC	DES encryption	597	481,779	537	0	17.4
DES DEC	DES decryption	525	423,675	537	0	15.5
RGB2YUV	Color Space Converter	104	83,928	116	0	3.8
2D DCT	2D Inverse Discrete Cosine Transform	314	253,398	191	0	11.5
QUANT	Quantization	125	100,875	124	0	8.7
Block RAM	On chip cache	26	20,982	24	32	25.0
Peripherals	UART, interrupt,timer	853	688,371	773	0	0.9
In Total		7675	6193,725	5295	38	

App	Architecture Description	Power(mW)	Time(ns)	Energy(pJ)
Adder	Microblaze+Adder+FSL+Cache+Peripherals	39.1	8.5	331.1
IDCT	Microblaze+IDCT+FSL+Cache+Peripherals	51.9	16.0	830.7
JPEG	Microblaze+ RGB2YUV +IDCT+Q+ FSL+Cache+Peripherals	61.0	31.4	1912.4
AES	Microblaze+AES_ENC+AES_DEC+FSL+Cache+Peripherals	71.7	39.8	2852.1
DES	Microblaze+DES_ENC+DES_DEC+FSL+Cache+Peripherals	70.0	25.1	1756.0

300x more energy than the hardware arbitration. Similarly, Fig. 13 also presents the evaluation metrics on different task scales. The speedup increase from 1.3x to 2.7x when the task scale increases from 64 to 4096, while the power consumption remains flat from 150.3x to 162.1x. The energy cost also increases from 204.8x to 427.6x accordingly. Above all, experimental results for both cases demonstrate the SoSoC architecture can improve the speedup as well as save significant proportion of power/energy consumption in a scalable manner.

5.6 Hardware Costs and Power Consumption

For the prototype system, we integrated 2 MBs, 1 adder module, 1 AES encoder, 1 AES decoder, 1 DES encoder, 1 DES decoder, 1 JPEG (2D-DCT) modules, and other peripheral blocks. Of the 2 MBs, one is used for scheduling, and the other is used as a computing servant. The hardware cost of the implemented system was evaluated. By looking into the modules of the system, we obtained the area and power consumption for each module.

In Table 4 the MB processor is the area consuming part (21.5 percent). The servants (e.g. Data Accumulation and AES cores) take 0.14$\sim$1.28 mm^2 of the fabric, which represent 2.4 $\sim$ 21.3 percent of the total area, depending on the complexity of each servant. Consequently, the additional power consumption is 2.0 mW $\sim$ 17.8 mW, which takes 2.6 $\sim$23.2 percent of the SoSoC prototype. Beyond the hardware cost and area utilization, the power consumption of the system only takes 118.0 mW, which demonstrates that SoSoC costs moderate energy and power consumptions.

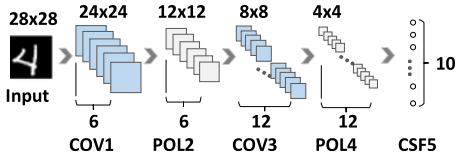


Fig. 14. Neural network hierarchy is containing different layers.

6 Convolutional Neural Networks as a Case Study

To demonstrate the efficiency of our SoSoC architecture in real applications, we use one typical case study, convolutional neural networks (CNN) in deep learning. Deep Learning has recently gained great popularity in the machine learning community due to their potential in solving previously difficult learning problems. Even though Deep and Convolutional Neural Networks have diverse forms, they share similar properties that a generic description can be formalized. First, these algorithms consist of a large number of layers, which are normally executed in sequence so they can be implemented and evaluated separately. Second, each layer usually contains several sublayers called feature maps; we then use the terms input feature maps and output feature maps. Overall, there are three main kinds of layers: most of the hierarchy is composed of convolutional and pooling layers, and there is a classifier at the top of the network consisting of one or multiple layers. The role of convolutional layers is to apply one or several local filters to data from the input layer. Consequently, the connectivity between the input and output feature map is local. Consider the case where the input is an image, the convolution is a 2D transform between a subset of the input layer and a kernel of the same dimensions, as illustrated in Fig. 14. Specifically, we implement services for convolutional neural networks (CNN), using both Xilinx Zynq (integrated with ARM Cortex hard processor) and Altera DE5 (integrated with NIOS processor) development boards. Table 5 illustrates the comparison between our experimental results to the state-of-the-art literature.

Experimental results demonstrate that our implementation can achieve 7.7 GFLOPS using Xilinx Zynq board with 43200 slices+220 DSPs, and 12.95 GFLOPS using Altera DE5 board with 234,720 ALMs+256 DSPs. The major difference between the development boards is the number of DSP blocks. The Xilinx Zynq board and Altera DE5 board have only 220 and

TABLE 5
Hardware costs, area and power of SoSoC architecture (CNN Case Study)

Metrics	ICCD13 [30]	ASAP09 [31]	FPL09 [32]	FPL09 [32]	PACT10 [33]	ISCA10 [34]	Xilinx Zynq	Altera DE5
Precision	fixed point	16 bits fixed	48 bits fixed	48 bits fixed	fixed point	48 bits fixed	32 bits float	32 bits float
Frequency	150 MHz	115 MHz	125 MHz	125 MHz	125 MHz	200 MHz	100 MHz	200 MHz
FPGA chip	Virtex6 VLX240T	Virtex5 LX330T	Spartan-3A DSP3400	Virtex4 SX35	Virtex5 SX240T	Virtex5 SX240T	Zynq Zedboard	Stratix V
FPGA Capacity	37,680 slices 768 DSP	51,840 slices 192 DSP	23,872 slices 126 DSP	15,360 slices 192 DSP	37,440 slices 1056 DSP	37,440 slices 1056 DSP	53,200 slices 220 DSP	234,720 ALMs 256 DSP
LUT type	6-input LUT	6-input LUT	4-input LUT	4-input LUT	6-input LUT	6-input LUT	6-input LUT	8-input LUT
CNN Size	2.74 GMAC	0.53 GMAC	0.26 GMAC	0.26 GMAC	0.53 GMAC	0.26 GMAC	0.8447 GFLOP	2.27GFLOP
Performance	8.5 GMACS 17 GOPS	3.37 GMACS 6.74 GOPS	2.6 GMACS 5.25 GOPS	2.6 GMACS 5.25 GOPS	3.5 GMACS 7.0 GOPS	8 GMACS 16 GOPS	7.7 GFLOPS 7.7 GOPS	12.95 GFLOPS 12.95 GOPS
Performance Density (Slice)	4.5E-04 GOPs/Slice	1.3E-04 GOPs/Slice	2.2E-04 GOPs/Slice	3.42E-04 GOPs/Slice	1.9E-04 GOPs/Slice	4.3E-04 GOPs/Slice	1.45E-4 GOPS/Slice	5.52E-5 GOPS/ALM
Performance Density(DSP)	2.2E-02 GOPS/DSP	3.51E-02 GOPS/DSP	4.17E-02 GOPS/DSP	2.73E-02 GOPS/DSP	6.63E-03 GOPS/DSP	1.52E-02 GOPS/DSP	3.5E-02 GOPS/DSP	5.05E-02 GOPS/DSP

256 DSPs respectively, while Xilinx Virtex-5/7 have 1056/2800 DSP blocks, which can optimize the matrix multiplication operations in CNN computation. Consequently, the performance density of our approach, especially for DSP resources, significantly outperforms the related studies.

7 Conclusions and Future Work

In this paper, we have introduced SOA into MPSoC design and proposed SoSoC, which consists of an application servant, a scheduling servant, multiple heterogeneous software and hardware computing servants. Through a well-defined programming interface and diverse computing resources, a specific application can be dynamically scheduled and offloaded to either soft or hardware computing servants at run-time. A prototype system of SoSoC has been implemented in a single FPGA chip. Evaluation and experimental results demonstrate that SoSoC can achieve great data parallelism with minimal componentization overhead and hardware costs. From the experimental results, we conclude that by integrating the SOA concept with MPSoC architecture design, different MPSoC prototype systems targeted at various applications can easily be constructed and utilized. SOA can usefully enhance flexibility. The reduction in design complexity can accelerate the process of prototype system construction and evaluation, and thus significantly shorten time to market.

There are numerous future directions worth pursuing. First, improved task partitioning and further adaptive mapping schemes are essential to support automatic task-level parallelization. Second, we also plan to study the out-of-order task execution paradigm, exploring the potential exploitation of parallelism in sequential programs. Finally, we feel that the SoSoC concepts should also be applied to clusters and supercomputing machines.

Acknowledgments

This work was supported by the National Science Foundation of China (No. 61379040), Anhui Provincial Natural Science Foundation (No.1608085QF12), CCF-Venustech Hongyan Research Initiative (No.CCF-VenustechRP1026002), Suzhou Research Foundation (No.SYG201625), Youth Innovation Promotion Association CAS (No. 2017497), and Fundamental Research Funds for the Central Universities (WK2150110003).

References

[1] T. Chen, et al., "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Archit. Support Programming Languages Operating Syst.*, 2014, p. 269–284.
[2] C. Wang, X. Li, and X. Zhou, "SODA: Software defined FPGA based accelerators for big data," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2015, pp. 884–887.
[3] C. Galuzzi and K. Bertels, "The instruction-set extension problem: a survey," in *Proc. 4th Int. Workshop Reconfigurable Comput. Archit. Tools Appl.*, pp. 209–220, 2008.
[4] C. Wang, et al., "SOMP: Service-oriented multi processors," in *Proc. IEEE Int. Conf. Services Comput.*, 2011, pp. 709–716.
[5] C. Wang, et al., "CaaS: Core as a service realizing hardware services on reconfigurable MPSoCs," in *Proc. 22nd Int. Conf. Field Programmable Logic Appl.*, 2012, pp. 495–498.
[6] B. Xiaoying, X. Dezheng, and D. Guilan, "Dynamic reconfigurable testing of service-oriented architecture," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, 2007, pp. 368–378.
[7] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.
[8] D. V. Thanh and I. Jørstad, "A Service-Oriented Architecture Framework for Mobile Services," in *Proc. Advanced Ind. Conf. Telecommun./Service Assurance Partial Intermittent Resources Conf./E-Learning Telecommun. Workshop*, 2005, pp. 65–70.
[9] M. K. HAKI and M. W. Forte, "Service oriented enterprise architecture framework in services (SERVICES-1)," in *Proc. 6th World Congr.*, pp. 391–398, 2010.
[10] I. M. Delamer and J. L. M. Lastra, "Service-oriented architecture for distributed publish/subscribe middleware in electronics production industrial informatics," *IEEE Trans. Ind. Informat.*, vol. 2, no. 4, pp. 281–294, Nov. 2006.
[11] Z. Jia, K. Daniel, and L. Shiyong, "Confucius: A scientific collaboration system using collaborative scientific workflows," in *Proc. IEEE Int. Conf. Web Services*, 2010, pp. 575–583.
[12] E. Lubbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. Embedded Comput. Syst.*, vol. 9, no. 1, pp. 1–33, 2009.
[13] W. Peck, et al., "Hthreads: A Computational Model for Reconfigurable Devices," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2006, pp. 1–4.

[14] D. Koch, C. Beckhoff, and J. Teich, "A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2009, pp. 253–256.
[15] K. Rupnow, K.D. Underwood, and K. Compton, "Scientific application demands on a reconfigurable functional unit interface," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, pp. 1–30, 2011.
[16] T. Martin, et al., *FlexCore: Utilizing Exposed Datapath Control for Efficient Computing*. Dordrecht, the Netherlands: Kluwer Academic Publishers, 2009, pp. 5–19.
[17] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1995, pp. 126–135.
[18] M. A. Watkins and D. H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture, in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2010, pp. 497–508.
[19] J. Wawrzynek, et al., "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, Mar./Apr. 2007.
[20] S. Vassiliadis, et al., "The MOLEN polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.
[21] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Proc. 12th Int. Conf. Archit. Support Programming Languages Operating Syst.*, 2006, pp. 325–335.
[22] H. Li, et al., "SWAP: Parallelization through algorithm substitution," *IEEE Micro*, vol. 32, no. 4, pp. 54–67, Jul.-Aug. 2012.
[23] M. Pericas, et al., "A flexible heterogeneous multi-core architecture," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Techn.*, 2007, pp. 13–14.
[24] J. Castrillon, et al., "Task management in MPSoCs: An ASIP approach," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. - Dig. Techn. Papers*, 2009, pp. 587–594.
[25] N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "An application development framework for ARISE reconfigurable processors," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 4, pp. 1–30, 2009.
[26] E. Salminen, et al., "Overview of bus-based system-on-chip interconnections," in *IEEE Int. Symp. Circuits Syst.*, 2002, pp. II-372–II-375.
[27] C. Wang, et al., "MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 2, pp. 1–26, 2013.
[28] C. Wang, et al., "Hardware implementation on FPGA for task-level parallel dataflow execution engine," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2303–2315, Aug. 2016.
[29] K. Ian and R. Jonathan, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in *Proc. 16th Int. ACM/SIGDA Symp. Field Programmable Gate Arrays*, 2008, pp. 149–158.
[30] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Des.*, 2013, pp. 13–19.
[31] M. Sankaradas, et al., "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst. Archit. Processors*, 2009, pp. 53–60.
[32] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2009, pp. 32–37.
[33] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 273–284.
[34] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, pp. 247–257, 2010.

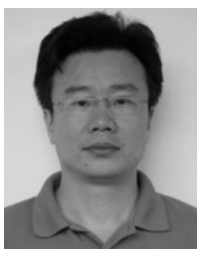
Chao Wang received the BS and PhD degrees from University of Science and Technology of China, in 2006 and 2011, respectively, both in of computer science. He is an associate professor in School of Computer Science, University of Science and Technology of China, Suzhou, China. He is the handling editor of Microprocessors & Microsystems, and IET Computers & Digital Techniques. His research interests focus on Multicore and reconfigurable computing.

Xi Li is a professor and vice dean in the School of Software Engineering, University of Science and Technology of China. There he directs the research programs in Embedded System Lab, examining various aspects of embedded system with the focus on performance, availability, flexibility and energy efficiency. He has lead several national key projects of CHINA, several national 863 projects and NSFC projects. He is a member of the ACM and the IEEE, a senior member of the CCF.

Yunji Chen received graduate degree from the Special Class for the Gifted Young, University of Science and Technology of China (USTC), Hefei, in 2002. Then, he received the PhD degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2007. He is currently a professor at ICT.

Youhui Zhang received the BSc and PhD degrees in computer science from Tsinghua University, China, in 1998 and 2002, respectively. He is currently a professor in the Department of Computer Science at Tsinghua University. His research interests include computer architecture and neuromorphic computing. He is a member of the IEEE and the IEEE Computer Society.

Oliver Diessel is an Associate Professor with School of Computer Science and Engineering, University of New South Wales, Australia. He is now the associate editor for ACM TRETS, General Chair for ICFPT 2009, Co-Chair for ASP-DAC 2012 Student Forum, TPC member of FCCM, FPL, FPT, ARC and RAW conferences, and a reviewer for many ACM/IEEE transactions and journals. His mainly research interest covers all aspects for reconfigurable computing systems, architectures, applications, algorithms, circuits. He has authored more than 60 papers on international journal and conferences.



Xuehai Zhou is a professor in the School of Computer Science, and the executive dean of School of Software Engineering, University of Science and Technology of China. He serves as general secretary of steering committee of computer College fundamental Lessons, and technical committee of Open Systems, CCF.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.