ELectronics EXpress

LETTER

IEICE Electronics Express, Vol.13, No.2, 1–9

# A MapReduce architecture for embedded multiprocessor system-on-chips

**Hao Xiao**[a)], **Huajuan Zhang, Fen Ge, and Ning Wu**

*College of Electronic and Information Engineering,*

*Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China*

a) *xiaohao@nuaa.edu.cn*

**Abstract:** With the advent of the Internet of Things, collection and processing of large datasets on embedded systems become increasingly important. Therefore, to enable embedded processors with more data processing capabilities, this paper presents a MapReduce-based multiprocessor system-on-chip (MPSoC) for providing efficient architectural supports to MapReduce parallel programming paradigm. We implement the proposed MPSoC in cycle-accurate SystemC and evaluate its performance using a set of representative MapReduce applications. Results show that the proposed MPSoC can achieve up to 2.1× overall performance improvement over the current general purpose multicore processors in typical MapReduce applications.

**Keywords:** multiprocessor system-on-chip, MapReduce, multiprocessor programming, embedded systems

**Classification:** Integrated circuits

## References

[1] W. Wolf and A. Jerraya: IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27** (2008) 1701. DOI:10.1109/TCAD.2008.923415

[2] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian and G. Reinman: ACM Trans. Embed. Comput. Syst. **13** (2014) 131. DOI:10.1145/2584664

[3] J. Castrillon, R. Leupers and G. Ascheid: IEEE Trans. Ind. Informat. **9** (2013) 527. DOI:10.1109/TII.2011.2173941

[4] M. Z. Urfianto, T. Isshiki, A. U. Khan, D. Li and H. Kunieda: IEICE Trans. Fund. Electron. Commun. Comput. Sci. **E91A** (2008) 1185. DOI:10.1093/ietfec/e91-a.4.1185

[5] H. Xiao, T. Isshiki, D. Li, H. Kunieda, Y. Nakase and S. Kimura: IPSJ Trans. Syst. LSI Des. Methodol. **5** (2012) 118. DOI:10.2197/ipsjtsldm.5.118

[6] H. Xiao, N. Wu, F. Ge, G. Zhu and L. Zhou: IEICE Trans. Inf. Syst. **E98D** (2015) 272. DOI:10.1587/transinf.2014RCL0001

[7] H. Xiao, N. Wu, F. Ge, T. Isshiki, H. Kunieda and J. Xu: accepted by IEEE Trans. Very Large Scale Integr. (VLSI) Syst. DOI:10.1109/TVLSI.2015.2408345

[8] J. Dean and S. Ghemawat: Commun. ACM **51** (2008) 107. DOI:10.1145/1327452.1327492

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis: IEEE Int. Symp. HPCA (2007) 13. DOI:10.1109/HPCA.2007.346181

[10] W. Fang, B. He, Q. Luo and N. K. Govindaraju: IEEE Trans. Parallel Distrib.



DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

Syst. **22** (2011) 608. DOI:10.1109/TPDS.2010.158
[11] M. M. Rafique, B. Rose, A. R. Butt and D. S. Nikolopoulos: IEEE IPDPS (2009) 5161062. DOI:10.1109/IPDPS.2009.5161062
[12] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu and H. Yang: ACM SIGDA Int. Symp. FPGA (2010) 93. DOI:10.1145/1723112.1723129

## 1 Introduction

With the advent of the Internet of Things, embedded devices tend to be more intelligent, making collection and processing of large datasets on embedded systems become increasingly important. Consequently, the demand for embedded processors that can provide more such capabilities under stringent cost and power constraints has greatly increased. Traditionally, application-specific integrated circuit (ASIC) is the common solution used to address such performance/energy challenge. However, with respect to programmable solutions, ASIC lacks reusability across different application domains, and significantly increases the overall design time and cost. Thus, a recent industry trend to address this design challenge is the use of programmable multiprocessor system-on-chip (MPSoC) [1, 2]. This solution, promising especially for its flexibility, is able to cover a wider spectrum of application domains and provide appealing time-to-market. In addition, it enables significant design space by allowing both coarse-grained thread-level parallelism and fine-grained instruction-level acceleration for achieving promising performance/energy tradeoffs.

However, due to the difficulties of parallel programming, managing the concurrency of MPSoC platforms is still not easy. Traditional parallel programming models, such as message-passing and shared-memory threads, requires the programmer to manually create threads and synchronize them through messages or locks, which is a very time-consuming and error-prone process. Although some recent works propose tools to partition sequential C/C++ programs to parallel ones [3, 4], developers still need to manually arrange the workload allocation and design the underlying inter-processor communication infrastructure. Our prior work proposed a software/hardware solution that can facilitate the MPSoC designer to handle the inter-processor communication [5, 6, 7]. However, it still lacks a general parallel programming model and a unified architecture which can help the designer to explore the inherent concurrency of applications.

MapReduce is a successful programming framework created by Google for processing large data sets on data centers [8]. The main advantage of MapReduce framework is that it can greatly simplify parallel programming, and facilitate processing of terabytes on large clusters. Therefore, due to its appealing capability in parallel programming, a lot of research works have exploited the MapReduce model on various chip-level parallel computing platforms. For example, Phoenix [9], developed by Stanford University, is a shared-memory version of MapReduce targeted for multi-core and multiprocessor systems. It uses shared-memory threads to implement parallelism, and its runtime can schedule tasks dynamically across the available processors for achieving load balance. In addition, MapReduce frame-


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

works build on GPUs [10], Cell clusters [11], and FPGA [12] are also implemented respectively.

Unlike the above works that exploit MapReduce model on commercial multi-core processors, this work aims to ease the development of dataset processing engines by applying the MapReduce model to customized MPSoC platforms. The proposed solution still leverages the MapReduce programming model to specify the concurrency at a high level, while the unique point is that a novel MPSoC architecture is proposed to provide hardware supports to the MapReduce runtime, including data scheduling, processing and merging. The proposed MPSoC uses multiple simple RISC processing elements (PEs) to spawn parallel Map and Reduce tasks. A hybrid shared-memory and message-passing on-chip communication infrastructure is adopted to facilitate both data transfer and inter-PE control messages. To demonstrate the effectiveness of the proposed architecture, we compare it with a commercial multicore processor with Phoenix programming model. Results show that our proposed architecture not only simplifies the MPSoC programming significantly, but also achieves an appreciable performance speedup than the general-purpose multicore processor.

The rest of this brief is organized as follows. Section 2 presents the proposed MPSoC architecture, which is followed by the performance evaluation in Section 3. Finally, we conclude the paper in section 4.

## 2 Hardware architecture

### 2.1 Architecture overview

Fig. 1 shows an architectural overview of the proposed MPSoC, which consists of a cluster of PEs connected by a hierarchical interconnection fabric. This architecture is designed based on the same principles of MapReduce model, and consists of three main components: the front-end for splitting input data, multiple map-reduce block-pipes (MRBP) for data processing, and the back-end for merging the mapreduce results.

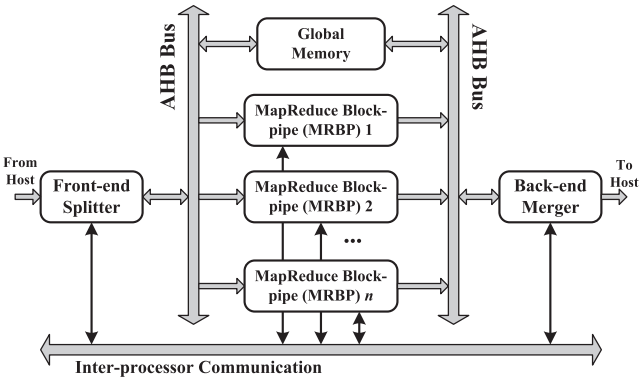


**Fig. 1.** Architecture overview of proposed MPSoC.

The front-end splitter is responsible for transferring the input data from the host system to MapReduce subsystem, as well as scheduling the following MRBP. Depending on the architectural configurations, it transfers the data to the global memory, dispatches small data blocks to each MRBP, and then triggers the


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

subsequent MRBP to start. MRBP is the kernel processing element of the architecture, which consists of a map-reduce pair connected by a shared AHB bus. According to the MapReduce model, it processes the input raw data into ⟨*key*, *values*⟩ pairs. Finally, after getting the results from each MRBP, the back-end subsystem starts to collect the separated ⟨*key*, *values*⟩ pairs to be a single entity and returns the final results.

For achieving fast block data transfer and easy implementation, the proposed architecture uses multiple AHB bus fabrics to connect the three components. Meanwhile, in order to relieve the data traffic on the bus, we separate the inter-processor control messages from the mapreduce data streams, and use another lightweight inter-processor connection to pass the control messages among the PEs. In the following subsections, the detailed architecture and data flow of the three subsystems will be explained.

### 2.2 Front-end subsystem

Fig. 2 describes the detailed architecture of the front-end subsystem, which consists of a splitter PE, a global memory and the data buffers of MRBP. The splitter PE is the kernel of this subsystem, which works as a direct memory access (DMA), as well as a central scheduler to control the MRBP. It is designed using our application specific instruction-set processor (ASIP) framework [5], whereby a communication module is integrated with a RISC PE to handle the data movement and inter-processor messages. As described in the blue dash lines, the splitter fetches the data from the host system and temporarily stores the data into the global memory. Then, depending on the architectural configuration, e.g., the number of MRBP, input data size, and data buffer size, the splitter further divides the input data into small blocks and dispatches these data blocks to the data buffer of MRBP. Finally, as described in the red dash lines, the splitter directly triggers the MRBR to start via inter-processor messages.

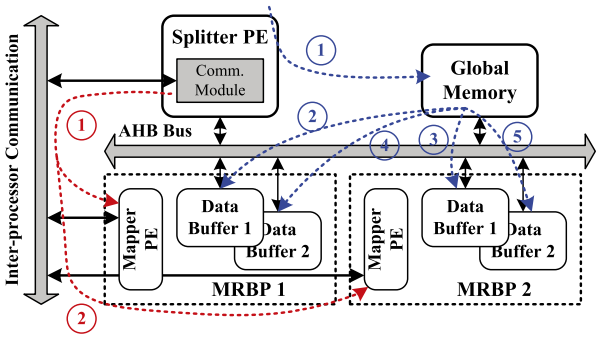


**Fig. 2.** Front-end subsystem: architecture and data flow.

In the front-end subsystem, the data movement, shown in the blue dash lines, goes though the traditional AHB bus. To relieve the traffic contention of the shared bus and achieve fast inter-processor communication, the control messages, shown in red dash lines, are passed via a lightweight point-to-point on-chip connection. Another tradeoff in the design is to use a single splitter or multiple splitters to boost the data dispatch. In this paper, we adopt the single splitter solution, because the


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

dispatch process takes much less time than the following map task. Thus, as long as there aren't too many MRBPs, a single splitter is enough to let all MRBPs start almost simultaneously. The splitter will dispatch data to MRBP whenever there is data buffer available.

### 2.3 MapReduce block-pipe

MRBP is the kernel data processing element of the architecture, which translates the input raw data into $\langle key, values \rangle$ pairs. As shown in Fig. 3(a), each MRBP consists of a map PE, a reduce PE and two ping-pong buffers, which are connected by a shared AHB bus. The ping-pong buffer structure is used to facilitate the data exchange between the map PE and the reduce PE. Moreover, the map PE and the reduce PE are also connected by the inter-processor communication network for passing control messages. As long as there is data available in the data buffer, either buffer 1 or buffer 2, the map PE is triggered by the front-end splitter to start processing. When map process finishes, reduce PE is triggered to further polish the data in the same buffer. Meanwhile, new data can be transferred into the other data buffer for map processing. In this way, the map and reduce PEs can work simultaneously in a block-pipe manner to improve the throughput.

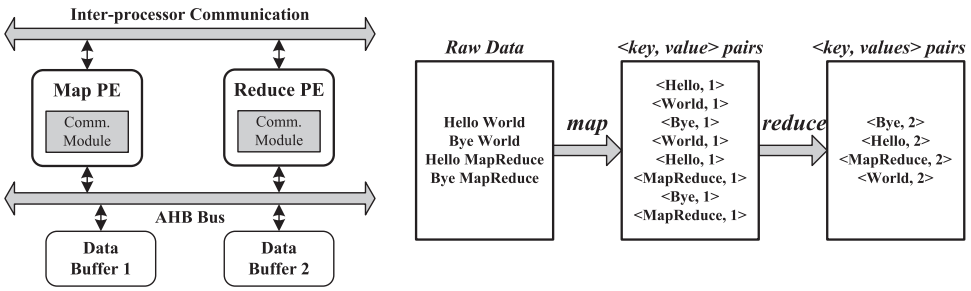


**Fig. 3.** (a) Architecture of a MapReduce block-pipe, (b) word count example using MapReduce model.

Fig. 3(b) shows the MapReduce processing flow with a word count example. According to the MapReduce model, the map PE first processes the raw data to be intermediate $\langle key, value \rangle$ pairs, and the reduce PE further polishes the intermediate $\langle key, value \rangle$ pairs to be sorted $\langle key, values \rangle$ pairs. Both map and reduce programs are user-defined depending on the target algorithms. The RISC cores (map PE and reduce PE) in the architecture provide a basic programmable platform for executing the map and reduce functions. In addition, designers can use our ASIP methodology [5] to further boost the map and reduce programs by exploring instruction-level accelerations.

### 2.4 Back-end subsystem

In the back-end subsystem, the reduced $\langle key, values \rangle$ pairs from all MRBPs (in the data buffers) are merged together into the global memory. In order to improve the throughput of merge processing, this design adopts a two-layered merge scheme. All MRBPs are equally divided into several groups and every merge


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

engine in the 1st layer takes over a specific MRBP group. In the 2nd layer, there is an additional merge engine that further combines all results from the 1st layer. Fig. 4(a) shows an example with four MRBPs, Merge 1 handles MRBP 1&2 and Merge 2 handles MRBP 3&4. Both Merge 1 and Merge 2 store their results into the global memory, and an additional Merge PE, in this example the Merge 3, further combines the results and obtains the final result. This two-layered merge scheme is implemented by using the multilayer AHB fabric, where the data buffers are private slaves to their first-layer merge PE, and the global memory is shared by all merge PEs. In addition, all the merge PEs are also connected via the global inter-processor communication network for passing control messages.

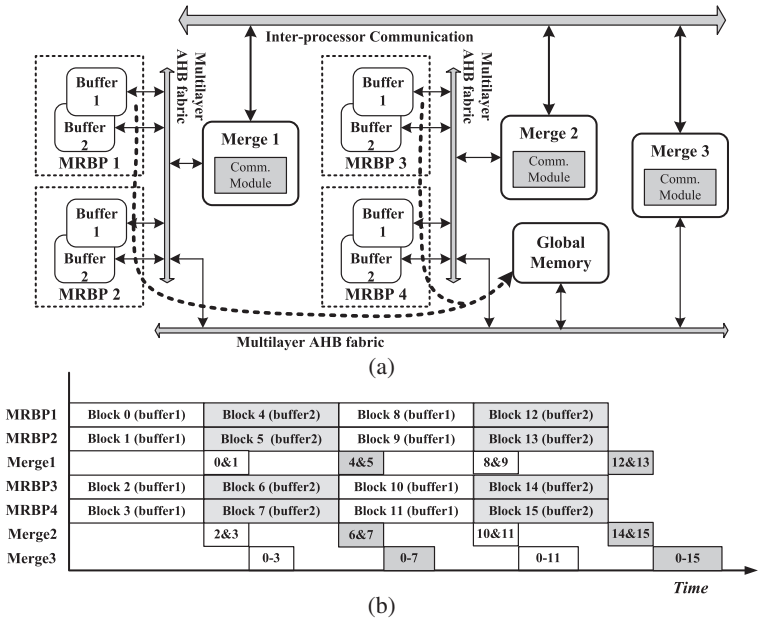


**Fig. 4.** Back-end subsystem: (a) architecture and (b) data scheme.

Fig. 4(b) shows a detailed data scheme of this subsystem. As explained in section 2.2, the splitting in front-end is very fast, and thus results from different MRBPs arrive almost simultaneously. However, compared with splitting, merging is more time-consuming. Therefore, using multiple merge engines to accelerate this process is necessary to keep the block-pipe working smoothly. It is noted that the processing time of the 2nd layer grows slightly when the results from the 1st layer accumulate. However, this does not affect the normal working of block-pipes, since the results from MRBP have been buffered in the global memory already. Moreover, depending on the target algorithms and data size, designers may tune the architecture with more merging layers for acceleration.

## 3 Implementation and results

### 3.1 System setup

The proposed MPSoC is implemented using our ASIP-based MPSoC design methodology [5], which tackles the hybrid shared-memory and message-passing architecture, the distributed interprocessor communication mechanism and the


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

ASIP design method. The full system is modeled in cycle-accurate systemC and the simulation is carried out using Synopsys Platform Architect. As listed in Table I, we evaluate 4 applications that have been implemented using the MapReduce programming model [9]. For comparison, we also implement the these benchmarks on Xilinx ZYNQ platform (ARM A9 dual-core processor) with Phoenix MapReduce framework. In addition, a single-thread implementation is also evaluated as a baseline reference. Table II lists the detailed architecture configurations.

**Table I.** The application used in this study

| | Description | Data Sets |
|---|---|---|
| Word Count | Determine frequency of words in a file | 50 MB, 100 MB, 200 MB, 500 MB, 1 GB |
| String Match | Search file with keys for an encrypted word | 50 MB, 100 MB, 200 MB, 500 MB, 1 GB |
| Linear Regresion | Compute the best fit line for a set of points | 50 MB, 100 MB, 200 MB, 500 MB, 1 GB |
| Histogram | Determine frequency of each RGB component in a set of images | 50 MB, 100 MB, 200 MB, 500 MB, 1 GB |

**Table II.** Processor architecture characteristics

| | ZYNQ | Proposed |
|---|---|---|
| Processor | ARM A9 | Customized MPSoC |
| Num. of Cores | 2 | 4 blockpipe |
| L1 Cache | 32 KB & 32 KB | None |
| L2 Cache | 512 KB | None |
| On-chip memory | 256 KB | 128 KB |
| Frequency | 667 MHz | 667 MHz |

### 3.2 Performance evaluation

Fig. 5 depicts the performance evaluation of 4 different tasks in terms of execution time. For each task, we use different data sizes (50 MB, 100 MB, 200 MB, 500 MB, and 1 GB). We measure three architecture configurations, which are single-core, dual-core with Phoenix MapReduce framework and the proposed MPSoC, respectively. As shown in Fig. 5, using the proposed MPSoC leads to a better performance than the other two architectures. Moreover, as the data size increases, the gained performance achieved by the proposed MPSoC also increases, which shows its potential capability in dealing with big datasets. Fig. 6 further shows the normalized execution time of 1 GB dataset. As shown, the proposed MPSoC is 1.7× to 3.0× faster than the single-thread implementation, while it is 1.5× to 2.1× faster than the dual-core phoenix implementation.


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

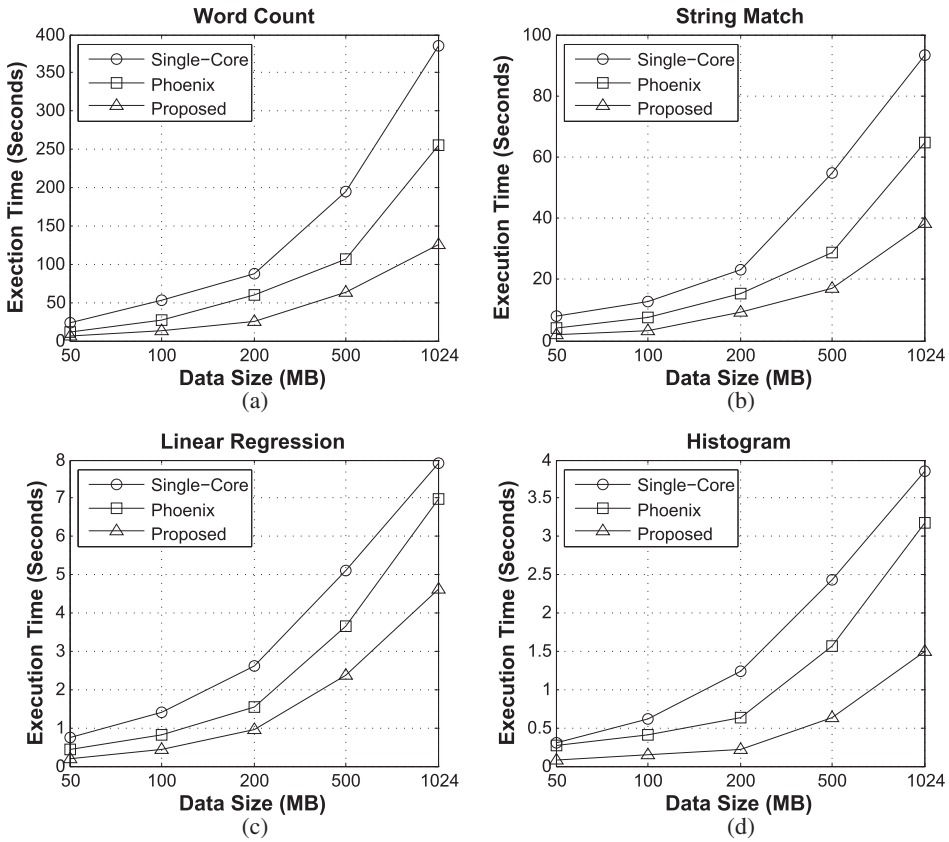


**Fig. 5.** Execution time for different applications.

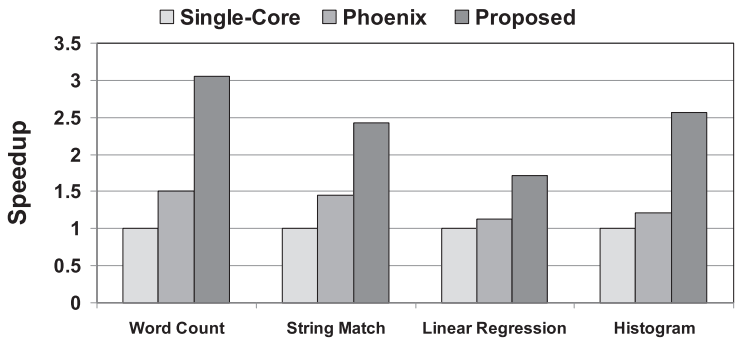


**Fig. 6.** Speedup for different applications using 1 GB dataset.

## 4 Conclusion

This paper proposes a MPSoC which provides architectural supports to the processing of large dataset using MapReduce parallel programming model. The proposed architecture uses multiple mapreduce block-pipes to spawn parallel Map and Reduce tasks, and adopts a two-layered merge scheme to improve the processing throughput. Moreover, a hybrid shared-memory and message-passing on-chip communication infrastructure is adopted, where the shared-memory scheme facilitates the data transfer between PEs, and the message-passing scheme is used for task scheduling. In the case study of a four-blockpipe MPSoC, we implement four typical mapreduce applications, and up to 2.1× speedup is achieved when compared with a commercial general-purpose multicore processor. In the future work,


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016

we would like to implement more realistic applications on this embedded MPSoC, and exploit distributed processing network built on top of this MPSoC.

## Acknowledgments

This work is supported in part by National Natural Science Foundation of China 61504059 61376025, Natural Science Foundation of Jiangsu Province BK20140834, the Industry-academic Joint Technological Innovations Fund Project of Jiangsu BY2013003-11, and Fundamental Research Funds for the Central Universities NS2015043.


DOI: 10.1587/elex.13.20151025
Received December 2, 2015
Accepted December 18, 2015
Publicized January 12, 2016
Copyedited January 25, 2016