ARTICLE IN PRESS

Computers and Electrical Engineering xxx (2011) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Computers and Electrical Engineering

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

An efficient scheduler of RTOS for multi/many-core system

Xiongli Gu [a], Peng Liu [a,*], Mei Yang [b], Jie Yang [a], Cheng Li [a], Qingdong Yao [a]

[a] Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China
[b] Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, United States

ARTICLE INFO

Article history:
Available online xxxx

ABSTRACT

Recently there is a trend to broaden the usage of lower-power embedded media processor core to build the future high-end computing machine or the supercomputer. However the embedded solution also faces the operating system (OS) design challenge which the thread invoking overhead is higher for fine-grained scientific workload, the message passing among threads is not managed efficiently enough and the OS does not provide convenient enough service for parallel programming. This paper presents a scheduler of master-slave real-time operating system (RTOS) to manage the thread running for the distributed multi/many-core system without shared memories. The proposed scheduler exploits the data-driven feature of scientific workloads to reduce the thread invoking overhead. And it also defines two protocols: (1) one is between the RTOS and application program, which is used to reduce the burden of parallel programming for the programmer; (2) another one is between the RTOS and networks-on-chip, which is used to manage the message passing among threads efficiently. The experimental results show that the proposed scheduler can manage the thread running with lower overhead and less storage requirement, thereby, improving the multi/many-core system performance.

1. Introduction

One of the major challenges faced by high-end computing machines or supercomputers, which are widely used in scientific computing area, is energy and power efficiency [1]. A promising way to improve the energy and power efficiency is to employ the low-power architecture developed for multi-core media processor in embedded computing [2]. In order to benefit from the rapidly improving computing power delivered by multi/many-core processors, software development plays an important role. Particularly, an efficient operating system (OS) is the key to better utilize the computing power provided by multi/many-core systems.

Our study is focused on the distributed multi/many-core system without shared memory and networks-on-chip (NoC) provides the on-chip interconnection architecture for such a system. The corresponding OS shall provide service for application programs, schedule the execution resources for efficient task/thread running and manage the message passing among processors with lower overhead on the interconnection network, as shown in Fig. 1. The major challenges [3] faced by designing such type of OS include: (1) the OS shall provide efficient protocol for the application programs to facilitate parallel programming and OS task/thread scheduling or invoking; (2) if the dynamic scheduling method is applied in parallel programming, the scheduling algorithm of OS shall guarantee the optimal system throughput at the cost of expensive context switch; (3) if the tasks/threads are mapped onto the processors statically at compile time, for fine-grained scientific workload, the OS shall invoke the task/thread for execution with less overhead to reduce the performance degradation which

* Corresponding author. Address: Department of Information Science and Electronic Engineering, Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang 310027, China. Tel./fax: +86 571 87953170.
E-mail address: liupeng@zju.edu.cn (P. Liu).

0045-7906/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2011.09.009

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

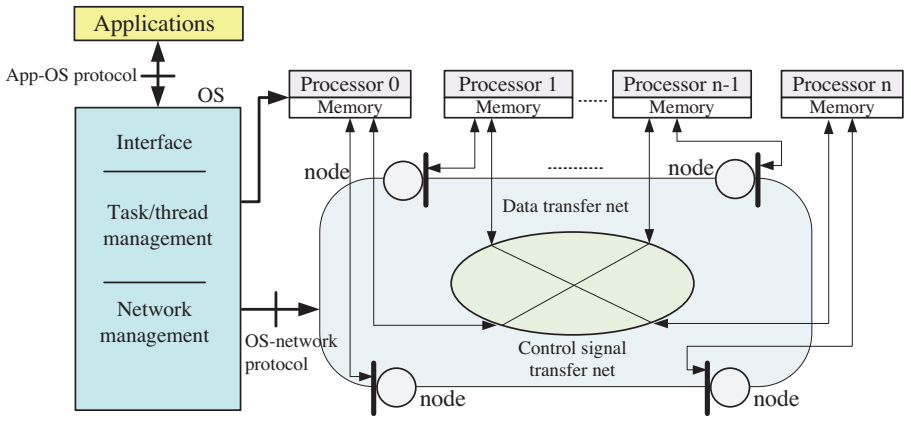


Fig. 1. An example OS for distributed multi/many-core systems without shared memory.

is caused by invoking overhead; (4) the management of message passing among processors is also important for fine-grained scientific workload, which has great effect on the system performance. The OS shall provide efficient protocol for the interconnection network in support of the convenience of message passing among processors.

The target applications are the data-driven [4] fine-grained scientific workload. The dataflow graphs (DFGs) [5] is applied to guide the parallel programming in our work. The partitioned tasks/threads are encapsulated in objects [6] which are statically mapped to the processors at compiler time. Thus, the OS needs not to dynamically schedule the objects to processors at runtime. The OS only invokes the object for execution followed by the atomic execution of the object. As a result, expensive context switch is eliminated. As such, in this paper the OS design mainly focuses on the following aspects: (1) providing an efficient protocol for the application program; (2) invoking the object for execution with lower overhead; (3) offering an efficient protocol for the interconnection network (NoC).

In this paper we propose a software scheduler, which is part of the master-slave RTOS, to efficiently manage the scientific workload running in multi/many-core systems. The master-slave RTOS is an extension version of our previous RTOS – Iota [7], as shown in Fig. 2. The RTOS includes the original Iota, a main scheduler and directors. The first two components are situated on the control processor, which are in charge of the synergistic work of multi/many cores. On each of the cores there is a director, also known as proxy for the scheduler, which is in charge of the management of the object execution in that core. During runtime all the directors work together to run the objects according to the arrangement of scheduler.

As we know, one trouble for multi/many-core system is the parallel programming – until now there has no effective way to completely solve this problem [8]. The proposed scheduler of RTOS defines an efficient protocol for the application program to release the burden of programmer, which the programmer only needs to provide the encapsulated objects and the data flow relation among them, and do not need to care their details of execution, such as data hazard problem in communication and synchronization problem. Instead, the proposed scheduler manages these operations. The data flow principle [9] is that any thread/object can fire (performing its computation) when (1) the input data on incoming buffer are available and (2) its output data buffer is also available. Thus after the scheduler on control processor dispatched the objects to

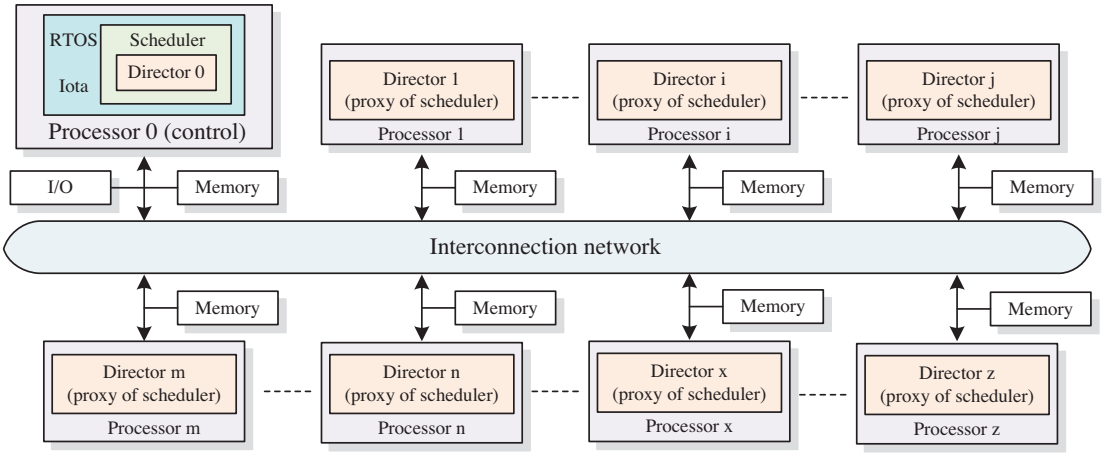


Fig. 2. A scheduler of RTOS situated in multi/many-core system.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

processors, the director on each core checks whether the input data of object are ready or not, invokes the ready object for execution and finishes the synchronization for the validity of output data buffer. The above processes are all based on message passing mechanism thus the control signal subnet of NoC, which specializes in transmitting the message (referring to control signal), and its corresponding functions in director are applied to accelerate the message passing among processors. The object is categorized according to the relation of one object to the other objects, which one object may have inter-processor and/or intra-processor predecessors/successors. As a result the object invoking flows are also classified according to object type in scheduler design. The above two methods efficiently reduce the object invoking overhead. The protocol between RTOS and NoC defines how RTOS accesses the network and the corresponding hardware constructs in NoC. The layered network and the corresponding functions in RTOS manage the message passing among objects efficiently. Furthermore, the scheduler of RTOS is tailored for multi/many-core system, which is with limited code size and small execution overhead to help improving system efficiency. Hence, the main contributions of this work include:

(1) The scheduler of RTOS defines an efficient protocol for the application program to amortize the difficulties in parallel programming.
(2) By message passing acceleration and object invoking flow categorization, the scheduler invokes the objects for execution with lower overhead than ever before.
(3) The efficient protocol between RTOS and NoC is defined in the scheduler design which manages the message passing among objects efficiently.

The rest of the paper is organized as follows: Section 2 presents an overview of the related work. Section 3 exposes the details of the proposed scheduler of RTOS Iota. Section 4 shows the evaluation methodology, the experimental environment and the analysis of the scientific kernels. Section 5 gives the experimental results and analysis. Finally the conclusions are made and future work is presented.

2. Related work

In the literature, a number of previous work for OS of multi/many-core system have been done, which address the OS design challenges in different aspects. In [10], the hardware scheduler *Carbon* is suggested to accelerate the task migration among processors for the fine-grained application programs in multi-core chip system. The hardware task management unit is proposed in [11] to find the ready threads for the following task/thread scheduling while the current thread is running. They both use hardware scheduler to reduce the task/thread scheduling overhead. However, the hardware scheduler lacks flexibility and its hardware cost will scale up with the number of the threads in the system. As for fine-grained scientific workload, numerous parallel threads will result in large hardware cost. In [12], the static scheduling is proposed for the synchronous data flow programs which eliminates the runtime scheduling overhead by scheduling parallel tasks/threads statically at compiler time. However, their method ignores the inter-processor communication overhead and the thread invoking overhead, both of which have great impact on system performance of fine-grained scientific workload. In our work, the parallel threads are encapsulated in objects and mapped to processors statically. The software scheduler is used to invoke the objects at runtime. This approach is more flexible and helps to reduce the thread invoking and communication overheads.

As for the protocol between OS and application program, in [13], the specification of sharing requirements for OS data, which is specified by application program, is suggested to improve the runtime system efficiency. In [14], the related thread id (RTID), which is provided by application program, is proposed to identify a collection of software threads to the scheduler in order to improve the runtime system performance. In this paper, the *DFG* is applied to partition an application program into many parallel threads which are statically scheduled to processors. The protocol is extended by providing the thread allocation information, their connection information, their priorities and data transfer information.

Some previous researches are focused on the communication protocols among processors. In [15] the NTU ICPC protocol, which cooperates with the hardware buffer, is proposed to reduce the data communication overhead among processors. However, for fine-grained scientific workload, data communication is more complicated. The communication protocols shall be modified to meet the requirement. In our previous work, the direct memory access (DMA) based protocol is suggested in OS design for data transmission among processors [6], which is also insufficient for the scientific workload. Other researches [16–18] are mainly concerned about the runtime OS scheduling algorithms which assure the maximum system throughput with expensive system cost.

3. Scheduler of RTOS Iota

3.1. Application model and execution model

Before describing the proposed scheduler, the application model and execution model are firstly introduced. The tasks in scientific workload will be executed once or iteratively according to threshold and each task will consume and produce certain amount of data. The tasks of a data-driven scientific workload can be modeled as a data flow graph.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

Definition 1. A *data flow graph (DFG)* [5] is a directed graph DFG(V, E, D), where each node $n_i \in V$ represents a task and a directed edge $e_k = (n_i n_j) \in E$ represents the communication between nodes n_i and n_j. Each edge $e \in E$ is associated with a nonnegative integer weight $D(e)$, representing a delay count between the iterations.

The data produced and consumed by node n in *DFG* can be specified a priori and *DFG* is an efficient representation of iterative computations like scientific workload. All tasks are executed in a self-timed manner [9] as follows: a task can be invoked if (1) it receives the data from all its predecessors and (2) its output data buffers are valid. Thus there are two types of buffer associated with each task: the input data buffers and the output data buffers: (1) the input data buffers are dedicated to receive data from its predecessors; (2) the task's produced data will be send to the output data buffers for the successors. In this paper, we assume that these buffers only exist in local memories of processors and the NoC just manages the data transmissions among them. Hence the synchronization for the buffers just happens among processors and the output data buffers of one task are just the input data buffers of its successors.

A task that needs to communicate with others is only allowed either at the beginning (i.e. read input data) or at the end (send produced data) of execution in one iteration. Thus the tasks are executed atomically and in this paper we define the atomic task as object.

By removing all edges of nonzero delays ($D(e) > 0$) from the graph, the *DFG* is converted to the *task graph* which extracts the iterative kernel from the application program [5]. Then the task scheduling algorithm can be applied to map the tasks to a finite set P of processors. In this paper, we assume that all the tasks have already been mapped to processors at compile time. We define A as the mapping of tasks V to processors P. Based on mapping A, we can get the execution graph.

Definition 2. An *execution graph* (*EG*) [19] is a directed graph EG(M, E), where each node $m_i \in M$ represents a mapped object associated with a message passing operation, and a directed edge $e_k = (m_i, m_j) \in E$ represents the *communication edge* (solid line) between processors or *program edge* between nodes (dashed line) that are within one processor, as shown in Fig 3. For object m_i,

- $w(m_i)$ gives the worst execution time of object m_i
- $f(m_i)$ gives the invoking overhead of object m_i

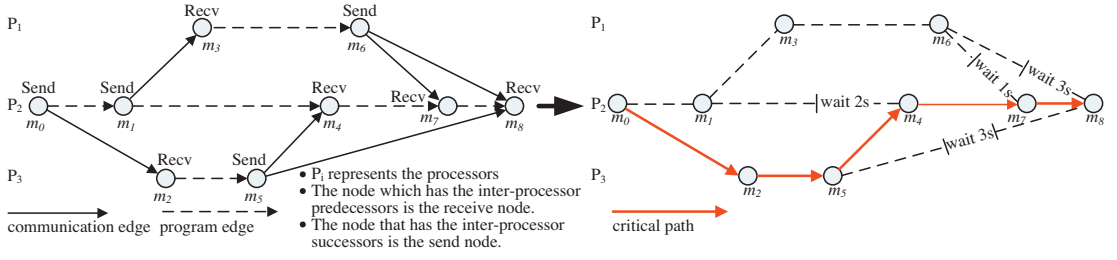
Each communication edge is associated with a communication delay $c(e_k)$ while the communication delay of program edge is zero for the memory sharing among intra-processor objects.

Definition 3. On the execution graph, the *critical path CP* is the longest path which determines the program's overall runtime of one iteration. It is the sum of communication delays of communication edge, the worst execution time and invoking overheads of object on the *CP*. The *CP* can be identified by the following two observations: (1) it cannot contain *program edges* leading to receive node that incurs a blocking wait time; (2) it cannot contain *communication edges* which lead to receive node that incurs a blocking wait time. Fig. 3 shows the critical path of the execution graph.

$$t = \sum_{j=1}^{N} \left\{ \sum_{m_i \in CP,M} w(m_i) + \sum_{m_i \in CP,M} f(m_i) + \sum_{e_k \in CP,E} c(e_k) \right\} \tag{1}$$

where the m_l represents the objects on the ***CP***, ϵ_k indicates the communications among objects on the ***CP*** and the constant N stands for the number of iterations. Eq. (1) shows that the program's overall runtime is determined by the three costs on the critical path. Thus the objective of scheduler design is to minimize t to obtain higher system performance.

As mentioned before, a task will be invoked if its input data and output data buffers are both available. The overhead of invoking a task mainly includes two parts (1) the overhead of checking the readiness of input data from its predecessors; (2) the overhead of synchronizing with its successors to confirm the availability of output data buffers. The predecessor/successor of one task may be mapped to the same or different processor that the task is mapped to. Hence, the process of checking



(a) Execution graph example (b) Critical path example

Fig. 3. An example of execution graph and critical path.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

the readiness of input data relies on the message sent from the processor that the task's predecessor is mapped to. So does the synchronization process with the processor that the successor is mapped. Thus the invoking overhead of one task is mainly decided by the efficiency of message passing and its corresponding invoking flow.

Another important feature in data-driven scientific workload is that the data transmissions between the processors are complicated. There are four types of data transmission: block data transmission, broadcasting transmission, word transmission and gathering transmission. The communication delay is mainly decided by the efficiency of managing the four types of data transmission.

In order to reduce the invoking overhead, realize efficient data transmission and amortize the difficulties in parallel programming, we propose to use a software scheduler which design is focused on the following three aspects:

- To reduce the invoking overhead, the categorized task invoking flow and efficient message passing (referring to the control signal, such as semaphore) mechanism are used.
- To realize efficient data transmission, the protocol between RTOS and NoC defines three subnets in NoC physical implementation and the corresponding functions in scheduler design.
- To amortize the difficulties in parallel programming, the protocol definition between application program and RTOS requires that the programmer only needs to supply the encapsulated objects and the data flow relation among them.

These will be discussed as below.

3.2. Protocol between application program and RTOS Iota

The protocol between the application program and RTOS specifies the parallel compiled representation of objects and their data flow relations, which can be mounted in the scheduler of RTOS and used by scheduler during the running of object. The definition of App-RTOS protocol (shown in Fig. 4) includes the following eight parts, which are listed in Table 1.

The protocol between application program and RTOS releases the burden of parallel programming so that the programmers can pay all their attentions on the parallel exploration of application programs and do not need to care the details of data hazard and synchronization problem in the execution phase.

3.3. Construction of scheduler and director

3.3.1. Construction of scheduler

As a functional part of RTOS Iota for the multi/many-core system, the scheduler with its proxy-director manages the synergistic working of tasks on different processors, as shown in Fig. 4. The scheduler handles the system initialization while the director manages the object execution at runtime. The scheduler is situated on the control processor core and the main structure is as follows:

(1) There is an interface between the application program and RTOS. The interface receives the objects and the connections between objects from the parallel compiler and fills them to the scheduler table. The interface also judges the type of the object based on the connection information and fills the type information to the scheduler table. As described in Section3.1, the processes of checking the input data readiness and synchronization shall depend on

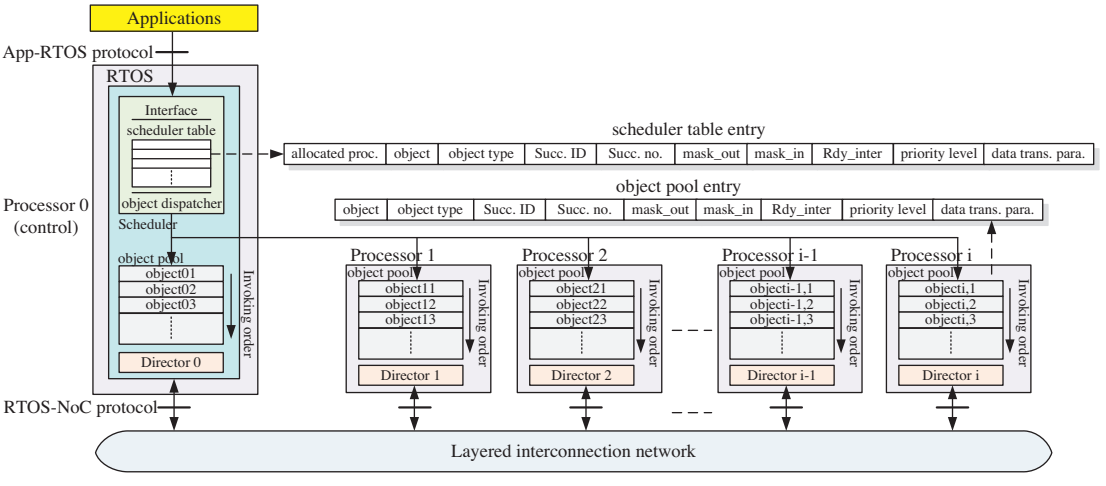


Fig. 4. Construction of scheduler and director.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

Table 1
Protocol between application program and RTOS Iota.

Element	Description
Object representation	Clarifying the object representation
Allocated processor	Indicating the allocated processor for the object
Successor ID	Representing the object's successors
Successor number	Indicating the number of successors
Maskin maskout	Clearing and setting the control signal (semaphore)
Ready_inter	Ready condition for inter-processor input data
Priority level	The invoking priority for object
Data transmission parameter	Clarifying the type and length of data transformation among inter-processor objects

the relation of one object to the other objects. This feature is exploited using the method of object categorization. Different invoking flows are suggested for the objects belonging to different categories, thereby, reducing the invoking overhead. Ten types of objects are defined and shown in Fig. 5. Table 2 shows the corresponding operations in invoking flow. Here the semaphore is applied as the control signal for synchronization between the objects.

(2) There is a scheduler table which contains the elements defined in App-RTOS protocol and the type information of object.
(3) There is an object dispatcher which dispatches the objects to the mapped processors at initialization stage based on the mapping information.
(4) As the crucial part of the scheduler, a director is situated on each processor including the control processor. A director is also known as a proxy of the scheduler. The director keeps a copy of the corresponding objects which shall be executed in that processor. The copy of connections for these objects either inner or outer of the processor shall be kept in the director as well. According to the copy the director manages the execution of the object on its situated processor.

3.3.2. Mechanism of message passing acceleration

For objects mapped within a processor, their communication need not use NoC since they share the memory of the processor. The semaphore is used to indicate a virtual transfer of data in which the output data of an object are transferred to the input buffers of its successors. The shared semaphore list is used to accelerate the intra-processor message passing. Fig. 6 shows an example of 32-bit semaphore list supporting the communication of five intra-processor objects. Each bit represents one input buffer for the corresponding object.

The message passing among the inter-processor objects is managed by the object scoreboard in the control signal subnet of NoC (introduced in Section 3.4). The structure of the object scoreboard is shown in Fig. 7. The object scoreboard records the readiness of input data for the inter-processor objects. As shown in Fig. 7, each core has its entry in the board which consists of the response part (16-bit width, in gray color in Fig. 7) and the setting parts (each with 2-bit width, in white color in Fig. 7). The object can write the corresponding bits in setting parts associated with the cores their successors are mapped to. Also the object can write the corresponding bits in its core's response part to inform its predecessor that the input data has been processed. Each core can read all the entries in the board so that the message passing among inter-processor objects is realized. As the object scoreboard records the dynamic status of message passing among cores, it can also be viewed as part of the scheduler. It is clear that by using the object scoreboard, the overhead of realizing the complicated protocol in normal

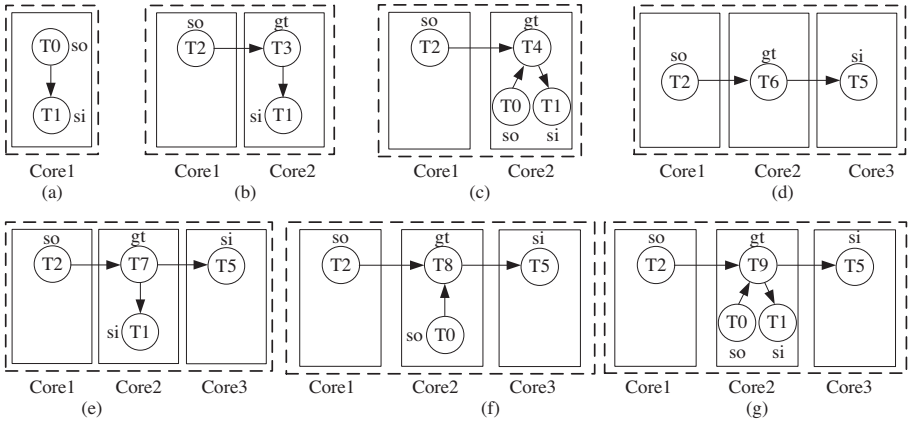


Fig. 5. Different types of object.

Table 2
Functions of the different invoking flows.

Type	Checking input data	Synchronization	Data communication
TO	None	Modifying shared semaphore	Output inner, no data transfer via NoC
T1	Checking shared semaphore	None	None
T2	None	Passing message among cores	Output via NoC to another core
T3	Checking passed message	Modifying shared semaphore	Output inner, no data transfer via NoC
T4	Checking shared semaphore Checking passed message	Modifying shared semaphore	Output inner, no data transfer via NoC
T5	Checking passed message	None	None
T6	Checking passed message	Passing message among cores	Output via NoC to another core
T7	Checking passed message	Modifying shared semaphore Passing message among cores	One output inner, no data transfer via NoC One output via NoC to another core
T8	Checking passed semaphore checking passed message	Passing message among cores	Output via NoC to another core
T9	Checking shared semaphore Checking passed message	Modifying shared semaphore Passing message among cores	One output inner, no data transfer via NoC One output via NoC to another core

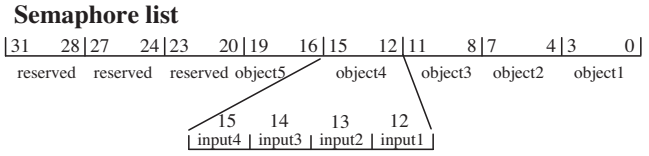


Fig. 6. Semaphore list example.

data transmission network is eliminated, thus the overhead of message passing (referring to the control signal) among the inter-processor objects is reduced obviously.

3.3.3. Structure and working flow of director

The director manages the operation of objects allocated in processor at runtime and the directors work together according to the scheduler table, which keeps the parallel objects executing synergistically. The main structure is as follows:

(1) There is an object pool which keeps part of the copy for the scheduler table. Each entry of the object pool records the object's information except the mapping information. The position of an object in the object pool indicates the invoking order of the allocated objects in that processor, which is decided by priority level defined in App-RTOS protocol – the object with higher priority shall be invoked firstly.
(2) The director manages the object running by using the following eight functions: *read_port*(), *response_set*(), *validation_check*(), *data_send*(), *message_send_inter*(), *response_clear*(), *message_send_intra*() and *clear_semaphore*(). The descriptions of these functions are shown in Table 3.

As described in Section3.1, the invoking flow of an object mainly includes two phases (1) checking the readiness of input data; (2) synchronizing with the successor, thus the director works as a finite-state machine, which has three states named object activation, object execution and synchronization check, as shown in Fig. 8.

(1) Object activation phase
In this phase the director checks the readiness of input data for the object which is currently in the head of the object pool. Different invoking flows are proposed for the different object types as: the object with no predecessors can be invoked at anytime and objects of other types shall be activated based on the semaphores in the shared semaphore list and/or the object scoreboard. By calling the function *read port*(), the director uses the round-robin inspection method to check the readiness of input data. The object which has inter-processor predecessor shall give its response signal after finding the readiness of the corresponding input data. The function *response_set*() is applied by director to modify the status of object scoreboard.
(2) Object execution phase
If the head object is ready to be invoked for execution, the director invokes the head object followed by the atomic execution of the object. After the execution of the object, the process turns back to the synchronization check phase of the director program.
(3) Synchronization check phase
After the invoked object finished its execution, the director shall synchronize with successors to confirm the availability of the output data buffer and the object with no successor skips this phase. The function *validation_check()* is

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

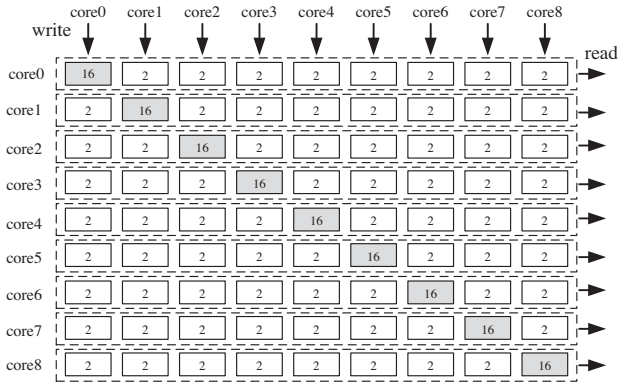


Fig. 7. Object scoreboard structure.

Table 3
Descriptions of the function.

Function	Description
read_port	Checking the readiness of input data
response set	Responding to the arrival of inter-processor predecessor's produced data
validation check	Synchronizing with successor to confirm validation of output data buffer
data send	Managing the data transmission among the inter-processor objects
message send inter	Sending message to validate the input data of inter-processor successor
response clear	Validating the output data buffer for the inter-processor predecessor
message send intra	Sending message to validate the input data of intra-processor successor
clear semaphore	Validating the output data buffer for the intra-processor predecessor

applied to confirm the validity of output data buffer. If the output data buffer is valid, the function *data_send()* is called to manage the data transmission among the inter-processor objects and the object with no inter-processor successor skips this operation. For synchronization and avoidance of data hazard, the function *message_send_inter()* and/or *message_send_intra()* set the semaphore for the successors and wait for the response signal of the inter-processor successor. When the response signal comes, the function *response_clear()* and/or *clear_semaphore()* will clear the bits set before by the function *response_set()* and *message_send_intra()*.

After all the above operations have been finished, the head pointer of the object pool moves to the next object and the director returns to the object activation phase. The director working flow for different types of objects are shown in Fig. 9.

3.4. Protocol between RTOS and NoC

The protocol between RTOS and NoC defines the software constructs in the scheduler of RTOS and hardware constructs in NoC and how they cooperate. As described in Section 3.1, there are four types of data transmission in the NoC of multi/many-core system. The NoC shall also transfer the control signal (semaphore) among processors. Three subnets are proposed in NoC and the corresponding functions are suggested in the scheduler design. The three subnets include conventional data transmission subnet, the word transmission subnet and the control signal subnet.

(1) A direct memory access (DMA) hardware is implemented in the conventional data transmission subnet for fast transfer of block, broadcasting and gathering data among the processor cores. The function *data_send*() in the director of RTOS manages the data transmissions in the DMA hardware, with the block size specified by the parameter of data transfer of the object.
(2) The multi-port register clusters are implemented as the medium for the fast word data transmission in the word transmission subnet. The function *data_send*() of director visits this subnet in direct memory-mapped accessing way to reduce the data transmission overhead. The first two subnets in the NoC and corresponding functions efficiently reduce the communication delay.
(3) Also the multi-port register clusters are implemented as the medium for fast control signal transmission in control signal subnet. The five functions *read_port*(), *response_set*(), *validation_check*(), *message_send_inter*() and *response_clear*() are applied in the director of RTOS to communicate with the object scoreboard in the control signal subnet of NoC. These functions visit this subnet in direct memory-mapped accessing way to enable a fast message transmission between the scheduler and NoC.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

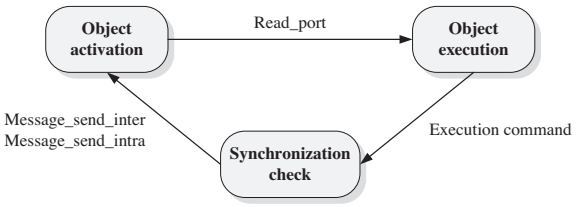
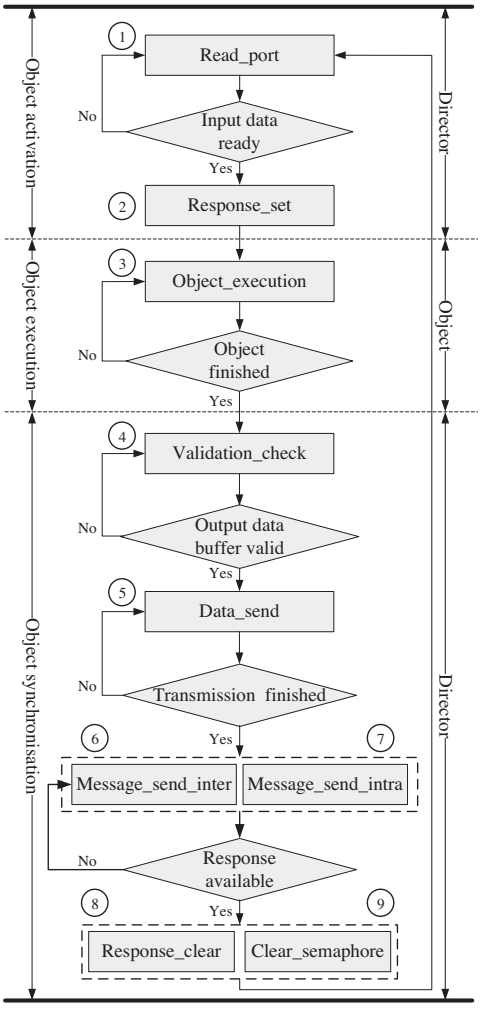


Fig. 8. Finite-state machine of director.



Object type	Invoking flow
type0	3,4,7
type1	1,3,9
type2	3,4,5,6
type3	1,2,3,4,7,8
type4	1,2,3,4,7,8,9
type5	1,2,3,8
type6	1,2,3,4,5,6,8
type7	1,2,3,4,5,6,7,8
type8	1,2,3,4,5,6,8,9
type9	1,2,3,4,5,6,7,8,9

Fig. 9. Director working flow diagram for different types of object.

4. Evaluation of the scheduler on fine-grained scientific workloads

4.1. Evaluation methodology and platform

As described before, the scheduler design mainly focuses on the following: (1) the categorized invoking flow for object management and the implementation of efficient message passing mechanism; (2) the definition of protocol between application program and RTOS; (3) the definition of protocol between RTOS and NoC. Hence, the experiments will be carried out as below: (a) the system performance improvement by the reduction of invoking overhead and data communication overhead will be evaluated and shown in experimental results; (b) the convenience of parallel programming brought by the App-RTOS protocol definition will be demonstrated in the test algorithm.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

Five key scientific computing kernels: matrix multiplication – Cannon's algorithm, the fast Fourier transform (FFT), eigenvalue of matrix – Jacobi algorithm, numerical solution of partial differential equation and LU decomposition algorithm are applied as the test algorithms. The *DFG* programming model guides the parallel programming for these application programs.

The evaluation platform integrates one 32-bit integer RISC core as the control CPU core and eight 32-bit integer DSPs as the computing acceleration cores. A 3×3 mesh topology conventional data subnet and a word transmission subnet are implemented for the data transmission among cores, and a control signal subnet is implemented for the control signal transmission among cores.

The RTOS Iota with its scheduler and the director of control CPU are allocated on the control RISC and the other directors are allocated on the DSPs. They work together to manage the object activation, synchronization and synergistic working for the system. The whole system has been designed at Register-Transfer Level (RTL) in Verilog HDL and the experiment is based on the RTL simulation.

4.2. Typical workloads in scientific computing

4.2.1. Cannon's algorithm

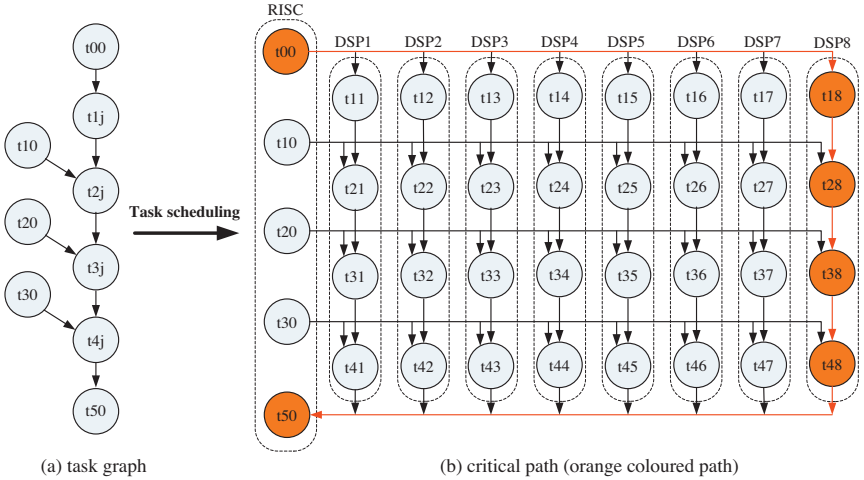
The matrix multiplication is used frequently in scientific computing. The Cannon's algorithm [20] is the most popular algorithm to solve the matrix multiplication. It separates the original matrix into the sub matrixes and each processor manages the sub matrix multiplication and addition. Then all the sub matrixes result on the processors form the final result. The $64 * 64$ matrix multiplication is applied as the test algorithm which is arranged to $4 * 8$ sub matrix multiplications and the sub matrix is of size $16 * 8$. Fig. 10 shows the parallel programming process and the critical path for Cannon's algorithm. As there is only one iteration in Cannon's algorithm, the DFG is the same as the task graph and in Fig. 10 only the task graph is drawn.

(1) The objects t_{i0} ($i = 0 \ldots 3$) are the predecessors for the inter-processor objects t_{ij} ($i = 1 \ldots 4$, $j = 1 \ldots 8$), which transfer the initial data to successors. They are allocated to the control RISC.
(2) The objects t_{ij} ($i = 1, 2, 3$, $j = 1 \ldots 8$) calculate the intermediate results for sub matrixes and they are the predecessors for intra-processor objects t_{i+1j} ($i = 1, 2, 3$, $j = 1 \ldots 8$). The objects are allocated to DSPs.
(3) The object t_{4j} ($j = 1 \ldots 8$) calculate the final results for the sub matrixes, which are the predecessors for the inter-processor object t_{50}. They are allocated to DSPs.
(4) The object t_{50} builds up the final matrix result and is allocated to the control RISC.

4.2.2. The fast Fourier transform (FFT)

The fast Fourier transform (FFT) is the fast algorithm for DFT. We use the FFT application program from the SPLASH-2 benchmarks suite [21]. In benchmark the Cooley–Turkey algorithm [22] is applied which is written as below:

$$X[k_1,k_2] = \sum_{n_2=1}^{N_2-1} \Bigg(\underbrace{W_N^{n_2k_1} \underbrace{\sum_{n_1=0}^{N_1-1} x[n_1,n_2] W_N^{n_1k_1}}_{N_1\text{-point DFT transfer}}}_{\bar{x}[n_2,k_1]} \Bigg) = \underbrace{\sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2k_2} \bar{x}[n_2,k_1]}_{PN_2\text{-point DFT transfer}}$$



(a) task graph (b) critical path (orange coloured path)

Fig. 10. Parallel programming process and the critical path of Cannon's algorithm.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

$$n = N_2 n_1 + n_2 \begin{cases} 0 \leqslant n_1 \leqslant N_1 - 1 \\ 0 \leqslant n_2 \leqslant N_2 - 1 \end{cases}, \quad k = k_1 + N_1 k_2 \begin{cases} 0 \leqslant k_1 \leqslant N_1 - 1 \\ 0 \leqslant k_2 \leqslant N_2 - 1 \end{cases}, \quad N = N_1 N_2. \tag{2}$$

In the experiment 4096-point FFT is applied as the test algorithm. The 4096-point FFT on platform is arranged as $N_1 = N_2 = 64$ and the Fig. 11 shows its parallel programming process and the critical path.

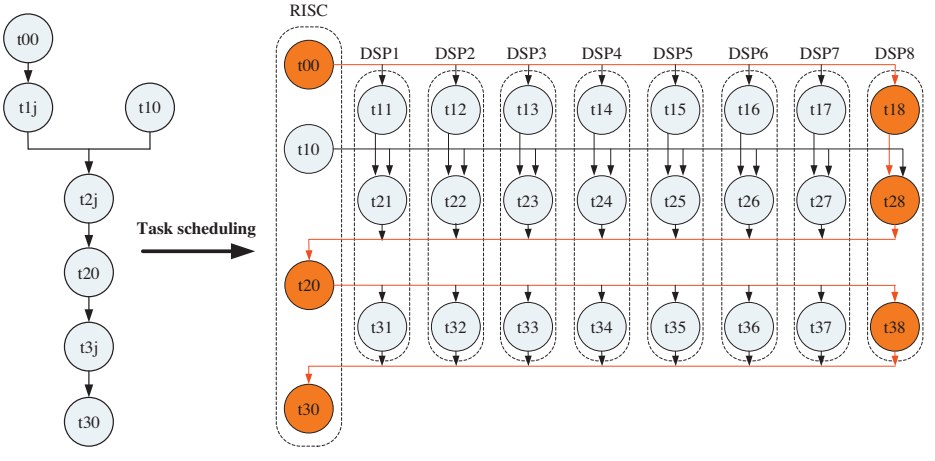
1. The object t_{00} is the predecessor for the inter-processor objects t_{1j} ($j = 1 \ldots 8$), which broadcasts the coefficients W_N and W_{N_1} to objects t_{1j} ($j = 1 \ldots 8$). It is allocated to the control RISC.
2. The object t_{10} is the predecessor for the inter-processor objects t_{2j} ($j = 1 \ldots 8$), which transposes the input data set, considered as $N_1 \times N_2$ complex matrix, into a $N_2 \times N_1$ matrix. It is allocated to the control RISC.
3. The objects t_{1j} ($j = 1 \ldots 8$) calculates the $W_N^{n_2}$ and $W_{N_1}^{k_1}$ for the intra-processor successor objects t_{2j} ($j = 1 \ldots 8$).
4. The objects t_{2j} ($j = 1 \ldots 8$) perform N_2 individual N_1-point one dimensional FFTs and multiply the results by $W_N^{n_2 k1}$. They are predecessors for the inter-processor objects t_{20} which are allocated to DSPs.
5. The object t_{20} transposes the resulting $N_2 \times N_2$ matrix into a $N_2 \times N_2$ matrix, which is the predecessor for the inter-processor objects t_{3j} ($j = 1 \ldots 8$) and is allocated to the control RISC.
6. The objects t_{3_j} ($j = 1 \ldots 8$) perform N_1 individual N_2-point one dimensional FFTs on the resulting $N_1 \times N_2$ matrix. They are the predecessors for the inter-processor successor t_{30} and are allocated to DSPs.
7. The object t_{30} transposes the resulting $N_2 \times N_1$ matrix into a $N_1 \times N_2$ matrix and is allocated to the control RISC.

We use the FFT algorithm as an example to demonstrate the convenience of parallel programming brought by the protocol definition between application program and RTOS. The object t_{i1} ($i = 1 \ldots 3$) and t_{30} are used as the samples which are shown in Table 4. The programmer only needs to provide the representation of object, the data relationship among threads and the attribute information of object, and does not need to care the data hazard problem and synchronization among the parallelism objects.

4.2.3. Eigenvalue of matrix

The eigenvalue λ of matrix **A** is defined as $\mathbf{Au} = \lambda\mathbf{u}$, where **A** is an $n \times n$ matrix, λ is a real number and $\boldsymbol{u}$ is the n dimensional characteristic vector of matrix $\boldsymbol{A}$. The Jacobi matrix algorithm [23] is applied to solve this problem. In experiment a $64 * 64$ matrix is applied as the test algorithm. The Fig. 12 shows its parallel programming process and the critical path of one iteration.

1. The object t_{00} assigns the matrix coefficients of **A** to the DSPs, which is the predecessor of objects t_{1j} ($j = 1 \ldots 8$). It is allocated to control RISC.
2. The object t_{1j} ($j = 1 \ldots 8$) divide the assigned coefficients into upper diagonal and lower diagonal classes, compute the biggest absolute value of matrix coefficients of the class that the assigned coefficients belong to and send the biggest absolute value and its row and column indexes to t_{20}. They are the predecessors of object t_{20} which are allocated to DSPs.
3. The thread t_{20} finds the maximum value from the collected biggest values, its row index g and column index h, and which class it belongs to as well. Then it decides if the whole coefficient except that of diagonal is not less than a threshold. If the result is true the iteration will be terminated and object t_{60} will get the final results. Oppositely the iteration will go to objects t_{3j}. The object t_{20} is the predecessor of objects t_{60} and t_{3j}, which is allocated to control RISC.



(a) task graph (b) critical path (orange coloured path)

Fig. 11. Parallel programming process and the critical path of FFT.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

Table 4
Information list provided by the application program.

Object	t11	t21	t31	t30
Allocated proc.	DSP1	DSP1	DSP1	RISC
Successor	t21	t20	t30	none
Successor no.	1	1	1	0
mask_out	0x00000010	0x00000000	0x00000000	0x00000000
mask_in	0xfffffff0	0xffffff0f	0xffffff0ff	0xffff0fff
ready_inter	0x00000001	0x00000002	0x00000004	0x88888888
Priority level	1	2	3	4
Data trans. para.	None	512 words	512 words	None

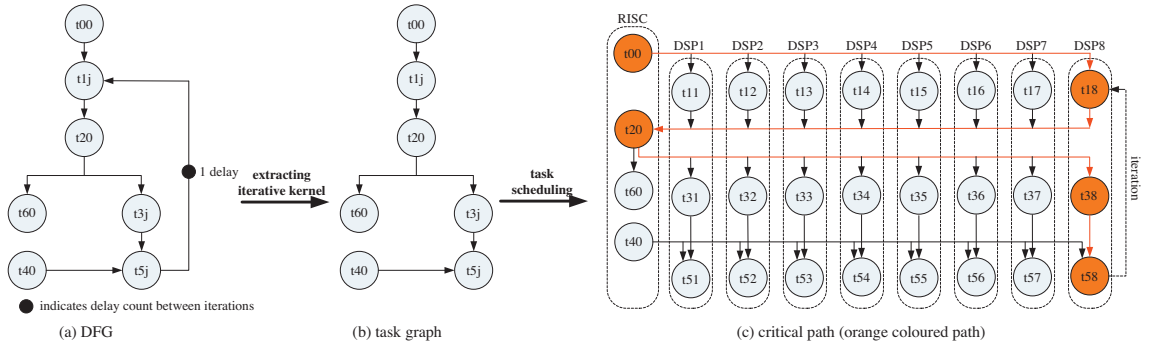


Fig. 12. Parallel programming process and the critical path of eigenvalue of matrix.

4. The objects t_{3j} ($j = 1 \ldots 8$) calculate the tangent of rotating angle θ of hyper plane (g, h), cos θ and sin θ, to form akin transform matrix $\mathbf{P}_{gh}$ and perform matrix multiplication $\mathbf{A}^{(k)} = \mathbf{P}_{\mathbf{gh}}^{(k-1)}\mathbf{A}^{(k-1)}\mathbf{P}_{\mathbf{gh}}^{(k-1)-1}$. They are the predecessors of objects t_{5j}, which are allocated to DSPs.
5. The object t_{40} broadcasts the matrix coefficients of row g and column h to the DSPs, which is the predecessor of objects t_{5j} ($j = 1 \ldots 8$). It is allocated to control RISC.
6. The objects t_{5j} ($j = 1 \ldots 8$) performs multiplication of the former result ($\mathbf{P}_{\mathbf{gh}}^{(1)}\mathbf{P}_{\mathbf{gh}}^{(2)}\mathbf{P}_{\mathbf{gh}}^{(k-2)}$) by $\mathbf{P}_{\mathbf{gh}}^{(k-1)}$, which are the predecessors of t_{1j} and allocated to DSPs. Because there is a loop between objects t_{5j} and t_{1j}, the initial status of t_{1j}'s input data buffer for t_{5j} shall be set valid to enable the running of objects t_{1j}.
7. The object t_{60} get the final results which the diagonal elements are the eigenvalues of the matrix and the columns are the corresponding eigenvectors. It is allocated to control RISC.

4.2.4. Numerical solution of partial differential equation

As an example the two-dimensional Poisson equation is defined as

$$\begin{cases} -\Delta u(x,y) = f(x,y) \\ u(x,y)|\partial\Omega = g(x,y) \end{cases} \quad (x,y)\epsilon\Omega = (0,W)\times(0,H) \tag{3}$$

$$u_{i,j}^{(n)} = \frac{h_x^2 h_y^2 f_{t,j} + h_y^2\left(u_{i-1,j}^{(n)} + u_{i-1,j}^{(n-1)}\right) + h_x^2 + h_x^2\left(u_{i,j-1}^{(n)} + u_{i,j-1}^{(n-1)}\right)}{2(h_x^2 + h_y^2)} \tag{4}$$

where $f(x, y)$ and $g(x, y)$ are the known functions and defined in the interior zone with boundary of Ω. The Gaussian–Seidal iteration method [24] is applied to solve this problem. In the experiment, the numeric computation of a specific partial differential equation is set on a 32×32 grid, which is divided into eight portions of size of 16×8 partial grids each assigned to one DSP. Actually, for the convenience of computation in a DSP the size of partial grid is enlarged to 18×10 for involving the data of border lines of samples from the neighboring partial grids, which leads the necessity of exchange of the border data from different partial grids after each iteration computing. The Fig. 13 shows its parallel programming process and the critical path of one iteration.

1. The object t_{00} distributes the data of boundary condition and the presumed initial data for the partial grids to DSPs. It is allocated to control RISC which is the predecessor of t_{1j}.
2. The objects t_{1j} ($j = 1 \ldots 8$) perform the calculation of expression (4), which are the predecessors of t_{20}. They are allocated to DSPs.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

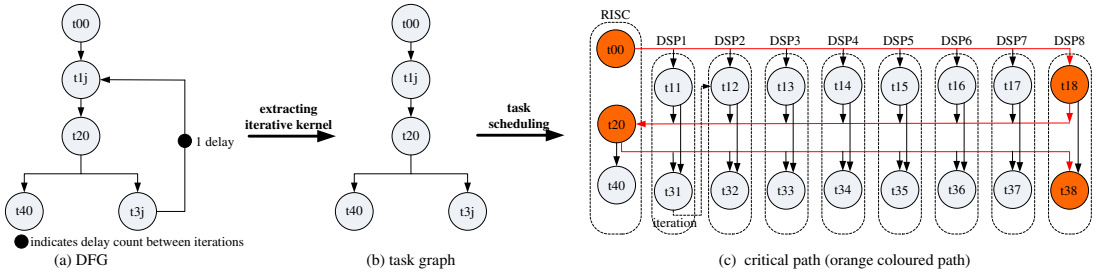


Fig. 13. Parallel programming process and the critical path of partial differential equation.

3. The object t_{20} compare the values from t_{1j} with the last iteration results whether the difference of them is less than a threshold or not. If the result is true the iteration will be terminated and object t_{40} will get the final results. Oppositely the iteration will go to objects t_{3j}. The object t_{20} is the predecessor of objects t_{40} and t_{3j} ($j = 1 \ldots 8$) which is allocated to control RISC.
4. The objects t_{3j} ($j = 1 \ldots 8$) perform exchange of border data with their neighboring objects, which are the predecessors of t_{1j} and are allocated to DSPs. As there is a loop between objects t_{3j} and t_{1j}, the initial status of t_{1j}'s input data buffer for t_{3j} shall be set valid to enable the running of objects t_{1j}.
5. The object t_{40} performs the output of the numeric solution of the partial differential equation, which is the successor of t_{20} and is allocated to control RISC.

4.2.5. LU decomposition algorithm

A system of n linear algebraic equations in n variables is usually expressed as $\mathbf{A}x = \mathbf{B}$,

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases} \tag{5}$$

where $\mathbf{A}$ is an $n \times n$ coefficient matrix containing the a_{ij}s, and x and $\mathbf{B}$ are n-element vectors storing x_is and b_is, respectively. The LU decomposition algorithm [24] is a well known solution to solve the linear algebraic equation which decomposes the coefficient matrix $\mathbf{A}$ into a product of a lower and an upper triangular matrix. The LU application program used in the experiment is from the SPLASH-2 benchmarks suite [21]. The 1024×1024 coefficient matrixes are applied in the experiment. Fig. 14 shows the parallel programming process and the critical path for LU decomposition algorithm.

1. The object t_{00} is the predecessor for the inter-processor object t_{11}, which arranges the input matrix $\mathbf{A}$ by using a 2-D scatter decomposition. It is allocated to the control RISC.
2. The object t_{11} is the predecessor for the inter-processor objects t_{2j} ($j = 2 \ldots 8$) and intra-processor object t_{21}, which processes the diagonal block of the matrix $\mathbf{A}$ and broadcasts the result to its successor objects. The objects t_{33}, t_{55} and t_{77} also do the same job as t_{11} does. They are allocated to DSPs.
3. The objects t_{2j} ($j = 1 \ldots 8$) process the blocks which are in the same row and column of the diagonal block and modify the blocks that are in the lower right corner of the matrix. They are the predecessors for the object t_{33} and are allocated on the DSPs. The objects t_{4j}, t_{6j}, t_{8j} ($j = 1 \ldots 8$) also do the same job as t_{2j} do.
4. The objects t_{01} collects the result data which is allocated to the control RISC.

5. Experimental results and analysis

As described in Section 4.1 the system performance improvements by the reductions of invoking overhead and data communication overhead will be evaluated and shown in experimental results. We compare the proposed scheduler of RTOS Iota with the former RTOS Iota (RTOS1) which is not optimized for reducing the invoking overhead and the data communication overhead. Table 5 and Table 6 list the three types of overhead on the critical path for the proposed scheduler and RTOS1. The percentage of the overhead to the whole time consumption, the system performance-speedup (the speedup to the single-processor) and the efficiency (the ratio of the valid acceleration cores to the eight acceleration cores) are calculated as the three criterions for the comparison. The results are shown in Fig. 15.

From Fig. 15 we can see that percentages of the invoking overhead in the proposed RTOS are lower which are no more than 11.51% and the invoking overhead of RTOS1 are larger than that in the proposed RTOS. As a result the RTOS1 has poor system performance than that of the proposed RTOS. The reductions of invoking overhead and data communication overhead improve the system efficiency by 0.13–19.62%. Thus the system efficiencies for the five algorithms are higher, which are not less than 59.00%.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

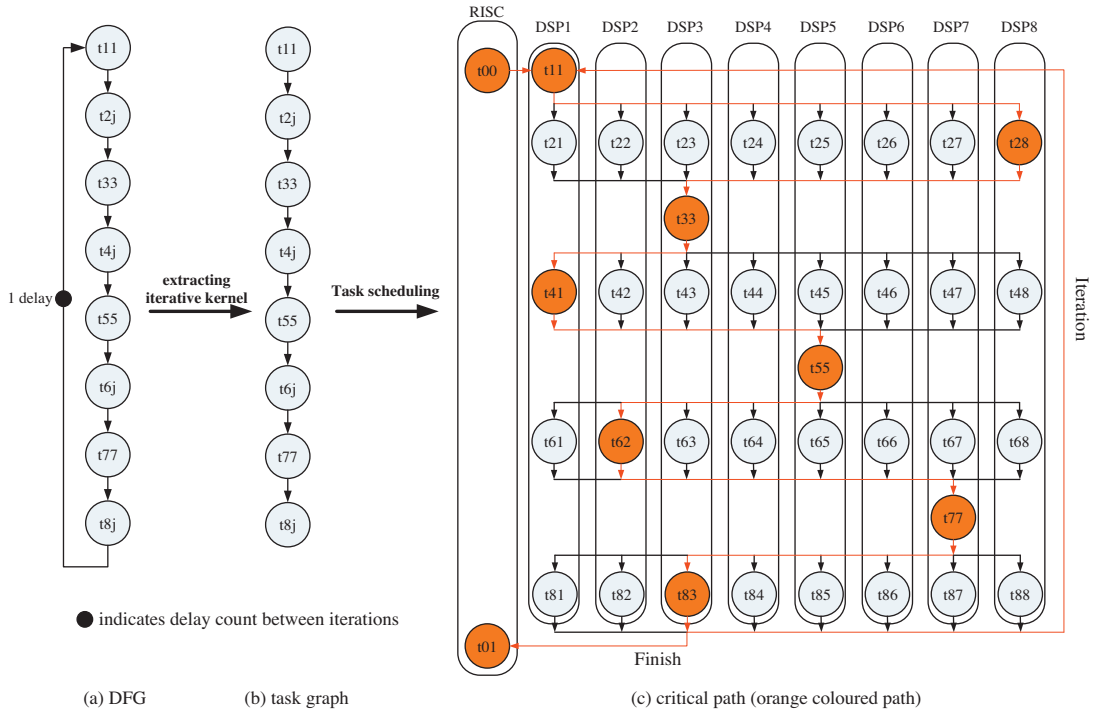


Fig. 14. Parallel programming process and the critical path of LU decomposition algorithm.

Table 5
Overheads of the five algorithms for the proposed scheduler of RTOS Iota.

Algorithm	Invoking (cycles)	Communication (cycles)	Computing (cycles)	Total (cycles)
Cannon	8421	44359	447614	500394
FFT	3448	16841	271725	292014
Eigenvalue of matrix	5633	6810	36502	48945
Differential equation	4161	1574	42703	48438
LU	7201	1886521	758044524	759938246

Table 6
Overheads of the five algorithms for the RTOS1.

Algorithm	Invoking (cycles)	Communication (cycles)	Computing (cycles)	Total (cycles)
Cannon	38337	44329	447614	530280
FFT	18648	17041	271725	307414
Eigenvalue of matrix	15870	11810	36502	64182
Differential equation	13032	6574	42703	62309
LU	78701	1886521	758044524	760009746

The data of efficiency of the cell processor taken from the literature show that, for 4096-point FFT, cell delivers 54.68% efficiency [25] while our system obtains 90.25%. And for the 1024 × 1024 LU algorithm, Cell's efficiency is 64.66% [26] while our system reaches 92.63%. Cell is a very good accelerator, which has been used in super computer Roadrunner [27], and the efficiencies in computation are higher than those are reported by [25,26]. We think that due to there is a very well designed operating system implementing in Roadrunner, which the computing efficiency of Cell is higher than that in [25,26]. On the other hand, the high computing efficiency of our multi core system just reveals that there is much rooms for the software/hardware co-design for the parallel programming with the operating system and the connection network between cores, which needs us to do further works.

The storage requirements of Iota/scheduler on RISC and directors on DSPs are estimated and listed in Table 7. From Table 7 we see that for the asymmetric structure of the RTOS, the storage requirements for Iota and director are small. It is very important for the limited memory space of the embedded processor cores.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

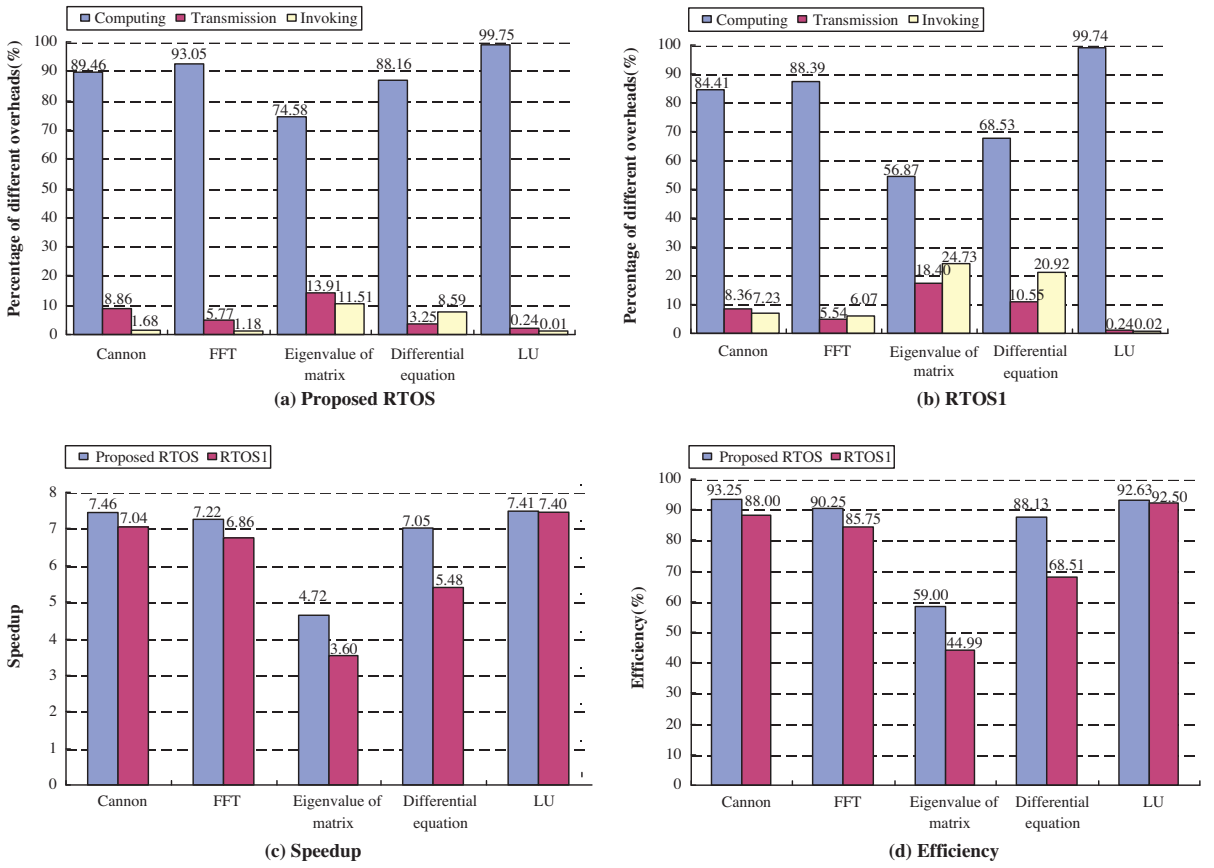


Fig. 15. Performance improvement.

Table 7
Storage requirements for Iota and director.

Memory space	Iota/scheduler on RISC (KB)	Director on DSPs (KB)
Operating system		
RTOS	10/3.660	5.380

6. Conclusions and future work

The suggested scheduler of master-slave RTOS presented in the paper offers an efficient way to manage the object running for the embedded distributed multi/many-core system without shared memories. It features of the small code size that the proxy of the scheduler of RTOS – director is only of size 5.380 KB, which is adequate for the limited memory space of embedded multi/many-core situation. By defining protocol between the application program and RTOS the suggest approach eases the difficulties of parallel programming, that enables the programmer to concentrate attentions on how to optimization his/her application without too much to concern the details on data conflicts or synchronization among the threads in the program. The experimental results indicate that the suggested scheduler and director and the clarification of protocol between RTOS and NoC improve the system efficiency by 0.13–19.62%, thereby, lead to higher system efficiency for the five scientific computing kernels. Our future work will be focused on the optimizations for the software structure of the proposed scheduler.

Acknowledgements

The authors thank the anonymous reviewers for their insightful comments, which helped us enhance the quality of our work. This work is supported in part by NSFC under grants 60873112 and 61028004, and the National High Technology Research and Development Program of China under Grant 2009AA01Z109.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009

References

[1] Torrellas J. Architecture for extreme-scale computing. IEEE Comput 2009;42(11):28–35.
[2] Donofrio D, Oliker L, Shalf J, Wehner MF, Rowen C, Krueger J, et al. Energy-efficient computing for extreme-scale science. IEEE Comput 2009;42(11):62–71.
[3] Manferdelli JL, Govindaraju NK, Crall C. Challenges and opportunities in many-core computing. Proc IEEE 2008;96(5):808–15.
[4] Treleaven PC, Brownbridge DR, Hopkins RP. Data driven and demand driven computer architecture. ACM Comput Surv (CSUR) 1981;14(1):94–143.
[5] Sinnen O. Task scheduling for parallel systems. Hoboken, New Jersey: John Wiley & Sons Press; 2007. p. 60–1.
[6] Cheng XM, Yao YB, Zhang YX, Liu P, Yao QD. An object oriented model scheduling for Media-SoC. J Electron (China) 2009;26(2):244–51.
[7] Gao F, Liu P, Yao QD, Li DX. Design and implementation of RTOS for a HDTV integrated source decoding chip. J Circuit Syst 2002;7(3):45–9.
[8] Patterson D. The Trouble with Multi-Core. IEEE Spectrum 2010;47(7):24–9.
[9] Bekooij M, Hoes R, Moreira O, Poplavko P, Pastrnak M, Mesman B, et al. Dataflow analysis for real-time embedded multiprocessor system design. Dynamic and robust streaming in and between connected consumer-electronic devices, vol. 3. Dordrecht: Springer Netherlands; 2006. p. 81–108.
[10] Kumar S, Hughes CJ, Nguyen A. Carbon: architecture support for fine-grained parallelism on chip multiprocessors. In: 34st Annual international symposium on computer architecture. San Diego, California; 2007. p. 162–73.
[11] Sjalander M, Terechko A. A look-ahead task management unit for embedded multi-core architectures. In: 11th Euro-micro conference on digital system design architecture, method and tools; Parma, Italy, 2008. p. 149–57.
[12] Lee E, Messerschmitt D. Synchronous dataflow. Proc IEEE 1987;75(9):235–45.
[13] Wickizer SB, Chen HB, Chen R. Corey: an operating system for many cores. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation; San Diego, CA, 2008. p. 43–57.
[14] Rajagopalan M, Lewis BT, Anderson TA. Thread scheduling for multi-core platforms. In: Proceeding of the 11th USENIX workshop on hot topics in operating systems; San Diego, CA, 2007. p. 2–7.
[15] Lin YH, Tu CH, Shih CS, Hung SH. Zero-buffer inter-core process communication protocol for heterogeneous multi-core platforms. In: 15th IEEE international conference on embedded and real-time computing systems and applications, Beijing, China, 2009. p. 69–78.
[16] Li T, Baumberger D, Koufaty DA, Hahn S. Efficient operating system scheduling for performance-asymmetric multi-core architecture. In: Proceeding of the 2007 ACM/IEEE conference on supercomputing, Reno, Nevada, 2007. p. 1–11.
[17] Li T, Brett P, Knauerhase R, Koufaty D, Reddy D, Hahn S. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In: 16th International symposium on high performance computer architecture (HPCA), India, Bangalore, 2010. p. 1–12.
[18] Sheng Y, Sheng YQ, Wei WX, Feng Z. Research on thread scheduling algorithm in automatic parallelization. In: International conference on computational intelligence and software engineering, Wuhan, China, 2009. p. 1–4.
[19] Schulz M. Extracting critical path graphs from MPI application. In: IEEE international cluster computing, Burlington, MA, 2005. p. 1–10.
[20] Cannon LE, A cellular computer to implement the Kalman Filter Algorithm 1969. Ph.D. thesis. Montana State University, Bozeman, Montana.
[21] Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd annual international symposium on computer architecture, New York, 1995. p. 1–13.
[22] Cooley JW, John WT. An algorithm for the machine calculation of complex Fourier series. Math Comput 1965;19(90):297–301.
[23] Golub GH, van der Vorst HA. Eigenvalue computation in the 20th century. J Comput Appl Math 2000;123(1/2):35–65.
[24] Press WH, Flannery BP, Teukolsky SA, Vetterling WT. Numerical recipes in C: the art of scientific computing. 2nd ed. Cambridge, England: Cambridge University Press; 1992. p. 34–42.
[25] Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K. The potential of the cell processor for scientific computing. In: Proceedings of the 3rd conference on computing frontiers, New York, 2006. p. 9–20.
[26] Venetis LE, Gao GR. Mapping the LU decomposition on a many-core architecture: challenges and solutions. In: Proceedings of the 6rd conference on computing frontiers, New York, 2009. p. 71–80.
[27] Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang M, Pakin, S, Sancho JC. Entering the petaflop era: the architecture and performance of Roadrunner. In: Proceedings of the ACM/IEEE conference on supercomputing, New York, 2008. p. 1–11.

Xiongli Gu received his BEng in Information Science and Electronic Engineering from Central South University, China, in 2005. He is currently pursuing his Ph.D. in Communication and Information System from Zhejiang University, China. His research focuses on high energy-efficient embedded processor micro-architecture, multiprocessor system-on-chip and parallel computer architecture.

Peng Liu received his BEng and MEng in Optical Engineering from Zhejiang University, in 1992 and 1996, respectively, and his Ph.D. in Communication and Electronic Engineering from Zhejiang University, China, in 1999. He has been an Associate Professor with the Information Science and Electronic Engineering Department, Zhejiang University, Hangzhou, China, since 2002. His research focuses on embedded processor design, multiprocessor system-on-chip architectures, on-chip interconnection networks, real-time operating system and parallel computer architecture.

Mei Yang received her Ph.D. in Computer Science from University of Texas at Dallas in 2003. She has been an Associate Professor in Department of Electrical and Computer Engineering, University of Nevada, Las Vegas. Her research interests include computer engineering, computer architectures, embedded systems, computer networks, wireless sensor networks and mobile ad-hoc networks.

Jie Yang received her BEng in Communication and Information System from Zhejiang University, China, in 2009. She is currently pursuing her MEng in Communication and Information System from Zhejiang University, China. Her research focuses on parallel programming and embedded system.

Cheng Li received his BEng in Communication and Information System from Sichuan University, China, in 2007, and his MEng in Communication and Electronic Engineering from Zhejiang University, China, in 2010. He is working at Mindray Company in Shenzhen, China. His research interests include embedded system and related software design.

Qingdong Yao is a Professor in Department of Information Science and Electronic Engineering, Zhejiang University, China. His research interests include multi-core system-on-chip, on-chip interconnection networks, high performance CPU and DSP design, communication, and video codec.

Please cite this article in press as: Gu X et al. An efficient scheduler of RTOS for multi/many-core system. Comput Electr Eng (2011), doi:10.1016/j.compeleceng.2011.09.009