PAPER Special Section on Parallel and Distributed Computing and Networking

Design and Implementation of a Contention-Aware Coscheduling Strategy on Multi-Programmed Heterogeneous Clusters

Jung-Lok YU[†a)], ***Nonmember and*** **Hee-Jung BYUN**[††b)], ***Member***

SUMMARY Coscheduling has been gained a resurgence of interest as an effective technique to enhance the performance of parallel applications in multi-programmed clusters. However, existing coscheduling schemes do not adequately handle priority boost conflicts, leading to significantly degraded performance. To address this problem, in our previous study, we devised a novel algorithm that reorders the scheduling sequence of conflicting processes based on the rescheduling latency of their correspondents in remote nodes. In this paper, we exhaustively explore the design issues and implementation details of our contention-aware coscheduling scheme over Myrinet-based cluster system. We also practically analyze the impact of various system parameters and job characteristics on the performance of all considered schemes on a heterogeneous Linux cluster using a generic coscheduling framework. The results show that our approach outperforms existing schemes (by up to 36.6% in avg. job response time), reducing both boost conflict ratio and overall message delay.

key words: *coscheduling, priority boost conflict, contention, runtime rescheduling latency, process reordering, myrinet, heterogeneous cluster*

1. Introduction

The design of efficient scheduler plays a crucial role in effectively utilizing underlying system resources. This becomes even more challenging on a general-purpose multi-programmed computing environments [1], [2]. Time-sharing is particularly attractive because it enables overlapped executions of processes with diverse demanding characteristics for shared resources [3]. The simplest approach to time-sharing a cluster is to leave each node to schedule its own processes autonomously. However, this form of scheduling can be very inefficient for parallel jobs that need process synchronization, mainly due to the lack of coordination among local schedulers [11].

Two main strategies for coordinating individual local schedulers have been proposed: *gang scheduling* (GS) [8], [14] and *communication-driven coscheduling* (CDC) [2]–[5], [11], [12], [15], [19]. GS uses explicit global synchronization to schedule all the processes of a job simultaneously. While GS is efficient for fine-grained parallel jobs, it can suffer from resource fragmentation when jobs are not well-balanced and/or do not make use of all nodes [17]. Recently, a full appreciation of these practical limitations of GS has led to a myriad of CDC schemes such as DCS [15], SB [5], PB [12], [19], and HYBRID [4]. These schemes use inherent communication events of parallel processes to approximately guide the local schedulers toward coscheduled execution. For example, on a message arrival, the implication is that the sender process is currently scheduled in a remote node. Thus, it will be of benefit to schedule or to keep scheduled the receiver process to reduce synchronization delay. Compared to GS, CDC schemes have better scalability and reliability, and have been shown to be efficient in clusters [4], [11]. Though these advantages, however, one major problem found in the existing CDC schemes is that they do not incorporate any steps to attempt to properly handle priority boost conflicts, leading to degraded performance.

To address this problem, in our previous study [6], we proposed a flexible CDC scheme that aims to efficiently resolve priority boost conflicts, and showed its feasibility on multi-programmed clusters. In this paper, we exhaustively explore the design issues and implementation details of our Contention-aware Coscheduling (CC) scheme over Myrinet [18]-based high performance Linux clusters. Note that unlike existing CDC schemes, CC adaptively rearranges the scheduling sequence of conflicting processes by exploiting the runtime difference in contention across remote nodes. To achieve this, each node measures its *rescheduling latency* - the approximated expected time when the current process is scheduled again on the node - as an index of contention at runtime, and piggybacks the information with normal outgoing messages. Based on the rescheduling latency notified from remote nodes, our scheme can grant more control over conflicting processes to the native scheduler so that it can schedule the process to be first whose correspondents will be scheduled sooner in remote nodes.

We also present a generic coscheduling framework that can be used to implement any CDC scheme with minimal effort, which is similar to [4], but more generalized and modulized so as to handle a variety of coscheduling strategies. We demonstrate the flexibility of this framework by implementing four prior representative CDC techniques (DCS, PB, SB, and HYBRID) and our CC schemes on real machines. Then, we thoroughly investigate the impact of various system parameters and job characteristics, such as job arrival patterns, multi-programming levels, and communication-intensity, etc., on the performance of CC and different CDC schemes. With the experimental results on a 16-nodes heterogeneous Linux cluster, we confirm that the proposed approach is very promising, especially for the hunting for wasted idle cycles in clusters.

Manuscript received December 28, 2010.
Manuscript revised May 31, 2011.

[†]The author is with Supercomputing Center, Korea Institute of Science and Technology Information, Republic of Korea.
[††]The author is with Suwon University, Republic of Korea.
a) E-mail: junglok.yu@kisti.re.kr
b) E-mail: heejungbyun@suwon.ac.kr

DOI: 10.1587/transinf.E94.D.2309

The rest of the paper is organized as follows. The next section provides a brief introduction of existing scheduling schemes, and Sect. 3 summarizes the basic concept of our Contention-aware Coscheduling. Section 4 describes the design and implementation details of our CC along with a unified coscheduling framework. Section 5 practically analyzes the performance implications of all considered CDC schemes. Finally, we conclude this paper in Sect. 6.

2. Coscheduling Strategies

2.1 Batch Scheduling (BATCH)

BATCH is the most popular policy to manage dedicated clusters for running non-interactive jobs. In BATCH, its Multi-Programming Level (MPL) is always one, which means that there is only one parallel job in a specific node in a cluster. The assigned job keeps running on the node until it is completed. LoadLeveler and PBS are widely-used batch schedulers. The disadvantages of BATCH are low processor utilization and high completion time [4]. To solve these problems, various techniques such as backfilling [9], etc., have been introduced. In our experiments, we use the OpenPBS [13].

2.2 Local Scheduling (LOCAL)

LOCAL is a baseline technique without the capacity to coschedule communicating processes. A receiving process spins until any message arrives, and is coscheduled with a sender process only if a message arrives while it is spinning. Though its simplicity, it yields unacceptable performance due to the lack of coordination among local schedulers.

2.3 Communication-Driven Coscheduling (CDC)

Communication-driven coscheduling (CDC) is a compromise of local scheduling (LOCAL) and gang scheduling (GS), and attempts to dynamically coschedule communicating processes to improve job performance using inherent communication behavior. As described in [4], [11], CDC schemes can be classified by two components: *message waiting action* taken by processes waiting for a message and *message handling action* performed by the kernel when a message arrives (see Table 1).

Table 1 Communication-driven coscheduling (CDC).

Scheme	Msg. Waiting Action		Msg. Handling Action (Boost Policy)
	Sender	Receiver	
LOCAL	Spin	Spin	Nothing
DCS	Spin	Spin	Interrupt & Boost
PB	Spin	Spin	Periodic Boost
IB	Spin	Im. Block	Interrupt & Boost
SB	Spin	Spin-Block	Interrupt & Boost
HYBRID	Im. Block	Im. Block	Boost & Restore (only collective comm.)

2.3.1 Spinning-Based Schemes: Dynamic CoScheduling (DCS) and Periodic Boost (PB)

Sobalvarro *et al.* [15] propose demand-based coscheduling (DCS) which uses incoming messages to schedule the processes for which they are intended. The underlying rationale is that the receipt of a message denotes the higher likelihood of the sender process of that job being scheduled at the remote note at that time.

DCS uses only spinning for sending and receiving a message. Periodically network interface card (NIC) has to get the ID of the thread currently executing on the host CPU. Since the NIC can not directly access host memory, this variable has to be DMAed onto the NIC's memory each time. On receipt of a message, NIC checks whether the intended destination of the message matches the estimated currently running process. If there is a mismatch, an interrupt is raised and the interrupt service routine (ISR) in the kernel is executed to boost the priority of the destination process to the highest value. This ensures that the destination process is scheduled soon after the receipt of a message.

Periodic Boost (PB) [12], [19] is an alternative coscheduling scheme to avoid expensive interrupt costs. In PB, the receiver is busy-waiting in send and receive operations, as with DCS; however, rather than raising an interrupt for each incoming message, a kernel thread periodically examines message queues of each process in a round-robin way and boosts the priority of one of the processes with unconsumed (or pending) messages. Whenever the scheduler is invoked in the near future, it would preempt the current process and schedule the boosted process. Several heuristics have been proposed to decide which process to boost at the thread invocation, but they only take the local states of processes with pending message(s) into consideration in selecting a candidate for a priority boost.

2.3.2 Blocking-Based Schemes: Spin Block (SB), Immediate Block (IB), and HYBRID

Unlike the former spinning-based schemes, blocking-based schemes use blocks as a message waiting action.

In Immediate Block (IB) [3], a receiver process blocks immediately if the expected message has not yet arrived, and is woken by the kernel when the message arrives.

Spin Block (SB) [5] is a compromise between spinning and blocking at the receiver side. A receiving process spins on a message arrival for a fixed amount of time, referred to as *spin time*, before blocking itself. The underlying concept of SB is that a process waiting for a message should receive it within the spin time if the sender process is also currently scheduled. Thus, if the message arrives within the spin time, the receiver process should hold onto the CPU to be coscheduled with the sender process. Otherwise, it should block and stop wasting the CPU resource. On the arrival of a subsequent message, NIC raises an interrupt that is serviced by the kernel to wake up the process and give a

priority boost to the awakened process.

As a variant of SB, HYBRID, recently proposed in [4], uses immediate-blocking for both senders and receivers to optimize spinning time. In addition, HYBRID explicitly boosts the priority of a process locally when it enters a collective communication phase (e.g., barrier, all-to-all), hoping that all other processes are also coscheduled, and restores its priority at the end of the phase.

3. Contention-Aware Coscheduling Approach

All the parallel jobs generally reveal different scheduling characteristics due to the load imbalance caused by multiprogramming and heterogeneity in cluster hardware resources. This leads to a dynamic change in the frequency of incoming and/or outgoing messages in each node, and this in turn can introduce serious obstacles to accurate coscheduling decisions. This problem is exacerbated in the real world, as parallel jobs themselves typically do not exhibit regularity in node requirements, communication patterns, etc. (i.e., application imbalance).

Priority boost conflicts normally occur on multiprogrammed clusters. Figure 1 depicts the occurrence of a boost conflict. Let P_{ij} denote the process of a parallel job i running on node N_j, while P_{i*} represents all the processes that belong to the same job i. m_{ij} is a message destined to P_{ij}. We assume that P_{xi} and P_{zk} enter their communication phases in N_i and N_k, respectively. We also assume that P_{yj} is currently scheduled on N_j, and that both P_{xj} and P_{zj} are blocked, waiting for a message. As depicted in the figure, the receipt of the new messages m_{xj} and m_{zj} during a short period of Δd in N_j results in the destined processes being awakened and boosted. However, at the next context switch (time t), it is not clear whether P_{xj} or P_{zj} should be scheduled first. Hence, the local scheduler has to make a choice between two or more candidate processes (P_{xj} and P_{zj} in this figure) for coscheduling with their correspondents - a boost conflict happens.

Scheduling conflicting processes should be carefully determined to optimize system utilization. Nevertheless, existing CDC schemes tend to make a simple decision when faced with a boost conflict. To illustrate this point, Fig. 2 (a) shows an example of a scheduling sequence of conflicting processes in existing CDC schemes. Note that because of multiple incoming messages (m_{1k}, m_{3k}, and m_{2k}) to node N_k, N_k has conflicting processes $\{P_{1k}, P_{3k}\}$ at time t and $\{P_{3k}, P_{2k}\}$ at t'. Let us assume that N_j is one of the nodes with many competing jobs (e.g., processes with pending messages) at time t, while N_m and N_l have fewer jobs than N_j. In existing CDC schemes, the local scheduler of N_k can easily generate the scheduling sequence of $P_{1k} \rightarrow P_{2k} \rightarrow P_{3k}$ for boost conflicts; however, this scheduling sequence encounters two major problems:

1. N_j already has numerous stringent processes with pending messages that are competing for its CPU. Thus, although a new message arrives from P_{1k} earlier and a priority boost is given to its destined process (P_{1j}), it takes a long time for P_{1j} to be rescheduled in node N_j. As a result, the probability of synchronization between P_{1k} and P_{1j} becomes low, causing additional excessive contention on the CPU in N_j.
2. On the contrary, in the worst case scenario the node N_m has no available work that can be run on its CPU (e.g., all processes are blocked or spinning while waiting for a new message arrival). In this situation, if N_k does not schedule P_{3k} immediately at time t, N_m continues with its idling state. This leads to low resource utilization.

The basic idea of CC is to dynamically regulate the scheduling sequence of conflicting processes by estimating the time when their correspondents are rescheduled in remote nodes. We call this *rescheduling latency*. Figure 2 (b) provides a conceptual depiction of CC. Note that, as in Fig. 2 (a), N_k has conflicting processes at time t and t'. As shown in Fig. 2 (b), if the local scheduler of N_k has information concerning the status of its peer nodes, it can schedule P_{3k} and P_{2k} at t and t', respectively, with the intention of not wasting CPU resource in N_m and N_l. Similarly, with this information, the local scheduler can delay the execution of P_{1k} to provide more time for N_j to schedule other boosted processes without intervention. In this way, by scheduling in advance one of the conflicting processes whose correspondents will be scheduled sooner in remote nodes, CC can not

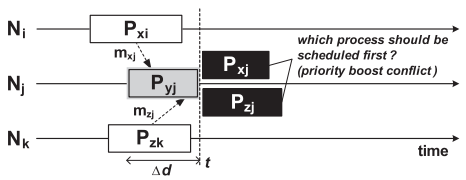


Fig. 1 Example of a priority boost conflict.

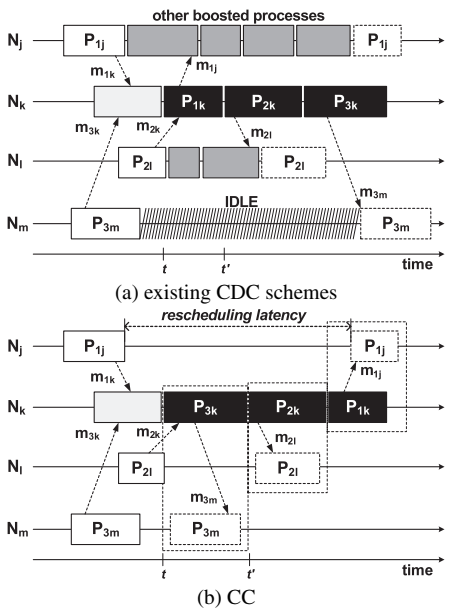


Fig. 2 An example scheduling sequence for conflicting processes.

only reduce unnecessarily wasted CPU time within the system but also increase the chance of synchronization among processes. This results in an overall improvement of system performance.

Note that the major difference between the proposed CC and the existing CDC schemes is that CC tries to guide coscheduled execution of parallel processes taking contention-imbalance across nodes into consideration on boost conflicts. PB selects a process with pending messages in a simple round-robin fashion on the same situation. DCS, SB, and IB boost the priority of processes based on the message arrival order, and HYBRID with preferring collectively communicating processes. However, these blind scheduling decision on resolving boost conflicts can lead to severe performance degradation, as described prior paragraphs. On the other hand, CC dynamically reorders the scheduling sequence of conflicting processes with the basis of rescheduling latency (i.e., a metric for contention-imbalance), hoping to minimize the wasted idle time in the system, as well as increase the synchronization ratio among processes.

4. Design and Implementation

4.1 Coscheduling Framework

To implement a coscheduling scheme, in general, it is needed to modify three major components: the NIC's device driver, the NIC's firmware, and the user-level communication library. A coscheduling algorithm is mainly implemented in the device driver (i.e., kernel level) to decide which process should be scheduled next (by boosting its priority). However, this traditional approach can require significant amount of effort and time. Moreover, this effort needs to be repeated whenever we move to another platform. Thus, we derived a coscheduling framework where a coscheduling heuristic can be cleanly abstracted as a stand-alone loadable kernel module. Note that it is similar to one in [4], but more generalized and modulized in that we present additional abstracted interfaces between kernel and loadable kernel modules (or device drivers).

Figure 3 shows our scalable framework used to implement various scheduling schemes. We provide a minimal kernel scheduler hook (i.e., abstracted interfaces) that allows the system software developers to change their local scheduling heuristics through an independently loadable kernel module. Then, we developed a coscheduling library and a dynamically loadable kernel module (named CoScheduling Module (CSM)), which support several reusable coscheduling policies. Every time the local Linux scheduler is invoked by the system, before making a selection from its own runqueue, the CSM selects the next process to run based on certain policy, and returns its recommendation to the native scheduler.

Using this framework, we have implemented prior coscheduling schemes (DCS, PB, SB, and HYBRID) and our CC. We also modified the GM driver and firmware to gather communication information used in different coscheduling schemes. As in [20], GM has one "Receive Event Queue" per an endpoint (*port*) for all notifications from a NIC; the send completion and receive completion event. To implement all schemes correctly, we divided the single event queue of GM into a "Send Completion Queue" (SCQ) and "Recv Completion Queue" (RCQ), and we only used the RCQ to detect whether a process has unconsumed messages or not for those schemes. For DCS, we added a minimal function to the NIC firmware that compares a current process to the destination process of the message and notifies the mismatch to the CSM by raising an interrupt. For HYBRID, to schedule a process differently depending on the computation and communication phases, we added the boosting/restoring code only in the collective communications of the Message Passing Interface (MPI) [10].

4.2 Implementation Details

The basic rationale of CC is that whenever a boost conflict is detected, the process whose corresponding processes will be scheduled sooner in remote nodes should be scheduled first (i.e., process reordering). Figure 4 depicts the overall structure involved in our CC implementation. The flexible coscheduling framework allows whether a process is subject to an alternate scheduling policy (DCS, PB, SB, HYBRID, and CC) or not. This functionality has been implemented in *cosched_lib*, and well-defined "reg-

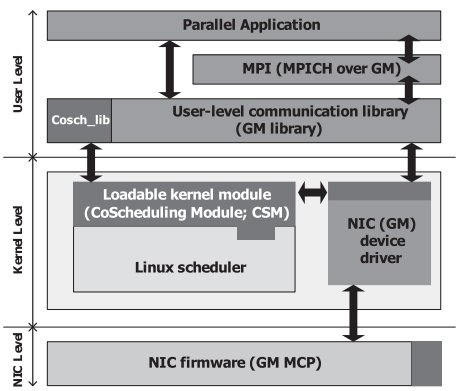


Fig. 3 A coscheduling framework.

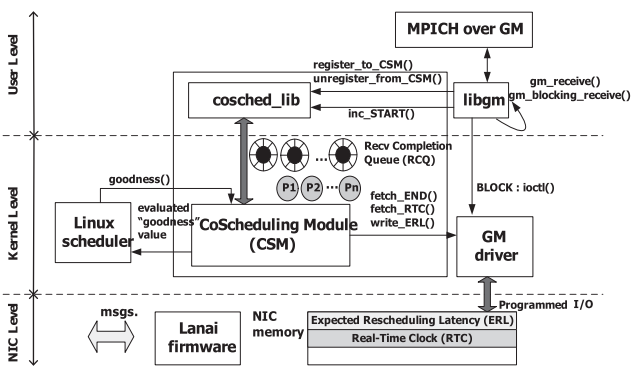


Fig. 4 Overall structure of CC implementation.

ister and unregister" interfaces (`register_to_CSM()` and `unregister_from_CSM()` as shown in Fig. 4) are provided to be invoked by the GM library†. During this "initialization" step, both "Send Completion Queue" (SCQ) and "Receive Completion Queue" (RCQ) (arbitrary memory pages used for send/recv notifications from NIC) belonged to the process are mapped to the kernel virtual memory by using `kmap()` function, and the START/END index of each queue is also mapped to the kernel memory area accessible by the CSM. This whole mapping mechanism enables the CSM to sniff meta-information (message length, etc.) on the send/recv completion events for the process registered to the CSM, as well as to detect if the process has pending messages or not.

CC is a complementary approach to existing CDC algorithms in that it only reorders the scheduling sequence of processes upon boost conflict. Therefore, CC can be easily integrated with existing schemes. As shown in Fig. 4, in CC-S (CC with spinning), both senders and receivers perform spinning (`gm_receive()`) waiting for the completion of send/recv operations, and process reordering is applied in a periodically invoked kernel thread (similar to PB). CC-B (CC with blocking) uses immediate-blocking (`gm_blocking_receive()`) at both sides if communications are not completed. Then, if a send or receive operation does not finish immediately, we block the process (`ioctl()`) and register for an interrupt with the NIC. As shown in Fig. 4, on a send completion/message arrival, if the process has registered itself with the firmware, the NIC interrupts the host to wakeup the blocked process. The woken-up process is placed in the ready queue, being a candidate for the next schedule. In CC-B, our reordering procedure is invoked at each context switch.

In our CC approach, whenever there exist the processes registered to the CSM, CSM updates the expected rescheduling latency (ERL) and calls the GM driver to download the measured ERL to the NIC at each context switch on its behalf (see `write_ERL()` function in Fig. 4). The driver makes a PIO call to make it available to NIC firmware. We approximately calculated the ERL value as the Eq (1):

$$\mathbf{ERL} = (\mathbf{ED} \cdot \mathbf{ENP}) + ((\lceil \mathbf{ENP} \rceil + 1) \cdot \mathbf{Cost_{switch}}) \quad (1)$$

where ED (Execution Duration) is the measured average CPU time (at granularity of μs) spent by each registered process between consecutive context switches, ENP (Expected # of Processes) is the number of processes with pending messages, and the context switching cost ($Cost_{switch}$) in Linux 2.4 kernel is about 6 μs [7]. Note that the ENP value is implicitly obtained from reordering function, as will be described in next paragraphs, by counting the process whose START and END index of RCQ has different value. The downloaded ERL value will be copied into the header of normal outgoing messages and transferred to other different nodes.

On a message arrival, the piggybacked ERL value extracted from the message and the timestamp (Real Time Clock (RTC) in NIC) for a corresponding process are DMAed onto an entry of RCQ pointed by END. At later, the process will consume the notified event of recv completion, making an increment of START index (*inc_START()* as shown in Fig. 4).

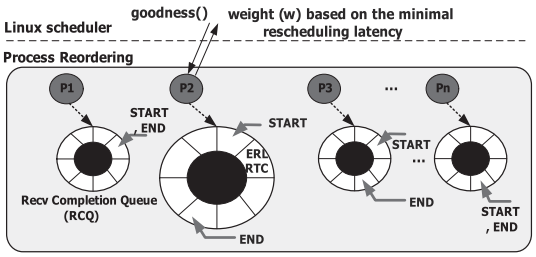


Fig. 5 The procedure of process reordering.

Every time the local scheduler gets invoked, it asks the CSM to recommend the next process to schedule (through *goodness()* function which is hooked by the CSM). For the evaluation of the weight for a candidate process, the CSM uses the mapped kernel memory area to access a list of the rescheduling latency information for the process as shown in Fig. 5. For each *goodness()* call for the process(es) in the ready queue, the total weight ($W_{total} = W_c + W_r$) of the process (P) is calculated in the proposed CC, according to the following simple Eqs. (2), (3), and (4):

$$\mathbf{W_c} = \begin{cases} 20 & \text{if the process has pending msg(s)} \\ 0 & \text{else if it has no pending msg(s)} \end{cases} \quad (2)$$

where 20 is constant weight factor to schedule first the processes with unconsumed messages, which mimics nice priority values (ranging from -20 to 20) in the Linux scheduler (In Linux, the default niceness for processes is inherited from its parent process, usually 0.). Then, for each pending message (m), we calculate:

$$\begin{aligned} \mathbf{\Delta t} &= \mathbf{fetch\_RTC}() - \mathrm{RTC} \\ \mathbf{RTC} &= \mathbf{RTC} + \mathbf{\Delta t} \\ \mathbf{ERL'} &= \mathbf{ERL} - \mathbf{\Delta t} \\ \mathbf{ERL} &= \mathbf{ERL'} \end{aligned} \quad (3)$$

Next, we find a proportional weight (W_r) for the process P,

$$\mathbf{W_r} = \frac{\min_{\forall m} \{\mathbf{ERL_{max}} - \mathbf{ERL'}\}}{\mathbf{NF}} \quad (4)$$

where ERL_{max}†† is the expected largest rescheduling latency in the system, NF is a normalization factor that adjusts W_r to the range of numerical values from 0 to 100. A constant weight (W_c) is assigned to a process(es) for which a message has already arrived, followed by a proportional weight (W_r) to all processes based on the minimal rescheduling latency of their correspondents in remote nodes. Adding the two

†Currently, in each node, maximum five number of processes can join to the CoScheduling Module (CSM) due to the limitation of available *ports* in GM.

††Multi-Programming Level (MPL) * 41 ms

weights (W_c + W_r) and recommending a candidate process with a higher total weight to the local scheduler provide us a reasonably accurate criteria to decide which process should be scheduled first in order to hunt wasted cycles in clusters.

In this way, we can establish a new improved scheduling order among conflicting processes, enforcing the local scheduler to select the most urgent process that communicates with the correspondents with the shorted rescheduling latency. Note that this is only one of the several possible reordering criteria, re-emphasizing its potential for further research direction.

5. Performance Analysis

In this section, we practically evaluate and analyze the performance of the CC schemes on real machines and compare it with those of others (LOCAL, BATCH, DCS, PB, SB, and HYBRID).

5.1 Experimental Setup

Our experimental testbed consists of one front-end node and 16-nodes Linux cluster (see Fig. 6). All computing nodes are connected through a 16-port Myrinet [18] switch for the data network, and further they are divided into two subgroups; seven computing nodes of Pentium-4 2.8 GHz processor and nine nodes of Pentium-4 1.8 GHz, each node with 512 MB memory and a PCI based intelligent Myrinet NIC, having 4 MB of on-chip RAM and a 133 MHz Lanai 9.2 RISC processor.

All computing nodes run Linux kernel version 2.4.32 with a minor scheduler hook. We modified the kernel by changing the default HZ value from 100 to 1000. This has a negligible effect on OS overhead, but makes the Linux scheduler re-evaluate process scheduling every ≈ 1 ms. As a result, considered all scheduling algorithms become more responsive, in particular, allowing us to control the frequency of periodic action used in DCS, PB, and CC-S being more fine-grained. We used Myrinet's GM implementation (version 1.6.5) over Myrinet's NIC as our user-level communication layer and MPICH over GM as our parallel programming library. GM supports two communication mechanisms; (i) non-blocking and (ii) blocking. In SB, HYBRID, and CC-B schemes, we use the blocking mechanism of GM.† On this platform, we measured application-to-application one-way latency to be around 13 μs by using *gm_allsize* micro-benchmark packed with GM distribution. This includes protocol processing overheads on both the sender and the receiver ends.

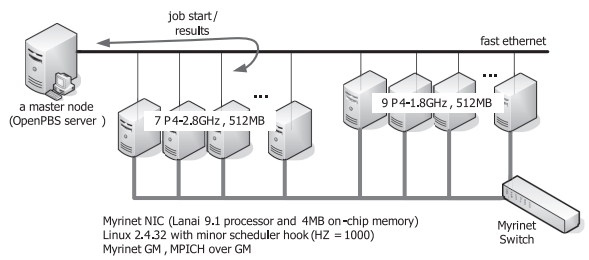


Fig. 6 Experimental setup (7 P4-2.8 GHz and 9 P4-1.8 GHz nodes).

5.2 Workload Characterizations

We use NAS Parallel Benchmarks (NPB, version 2.4) [21] to evaluate the performance of all scheduling schemes in this study. NAS benchmarks consist of a set of eight benchmark problems, and of these, we consider five applications with CLASS A: BT, SP, FT, IS, and CG; the lowest to highest communication intensities, respectively (see Table 2). BT and SP are two simulated CFD applications that solve multiple independent systems of block trigonal and scalar pentadiagonal equations. FT is a 3D partial differential equation solver using fast Fourier's transformation, which exchanges a large number of (66%) big message chunks. IS is an integer sorting application that uses a large number of small messages (< 4 KB). CG is a conjugate gradient matrix inversion algorithm with very heavy communication intensity (≈ 46%). The communications in benchmarks FT and IS are dominated by collective communications (i.e., All-to-All (AA) pattern), while the communications in CG, BT, and SP are dominated by point-to-point communications (i.e., Nearest Neighbor (NN) pattern).

Using a combination of above five applications, we designed a set of 6 parallel workloads (WL1 - WL6), varying in communication intensity and patterns. The complete workload space for which we have run the experiments is summarized in Table 3. All applications have been adjusted and compiled so they approximately take the same amount of time (66 - 73 sec) to complete, when executed individually on all 16-nodes. In a real system, a parallel job arrives at a cluster and waits in the waiting queue if it cannot be allocated immediately. After a certain amount of queuing time, it is assigned to the required number of processors.

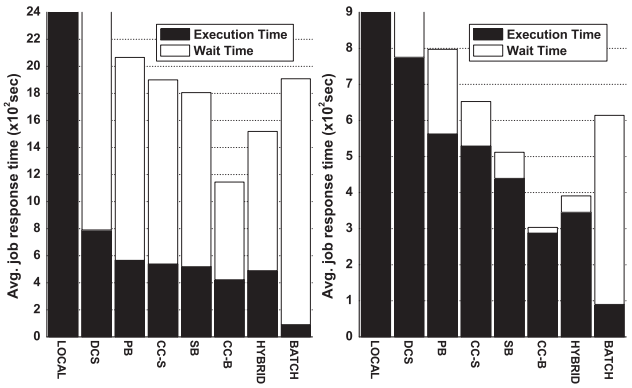
Table 2 Five applications from NPB (16 nodes, CLASS=A).

	Comm. Intensity	Comm. Pattern	Msg. Size Distribution	MPI Routines
BT	low	NN	11 K (37%) 61 K (26%) 69 K (37%)	MPI_Isend() MPI_Irecv() MPI_Wait()
SP	medium	NN	18 K - 20 K (37%) 39 K - 45 K (37%) 61 K (26%)	MPI_Isend() MPI_Irecv() MPI_Wait()
FT	medium	AA	8 (33%) 1048 K (66%)	MPI_Alltoall() MPI_Reduce()
IS	high	AA	4 (33%) 4 K (33%) 65 K (33%)	MPI_Allreduce() MPI_Alltoall() MPI_Alltoallv()
CG	high	NN	8 (56.5%) 16 (1.1%) 28 K (42.4%)	MPI_Send() MPI_IRecv() MPI_Wait()

†In SB, *spin time* for which a process spins on a message receive before blocking itself, is carefully chosen to optimize performance. We varied the *spin time* from 100-300 μs and we set it to be about 150 μs because it gives the best performance on the platform we are using.

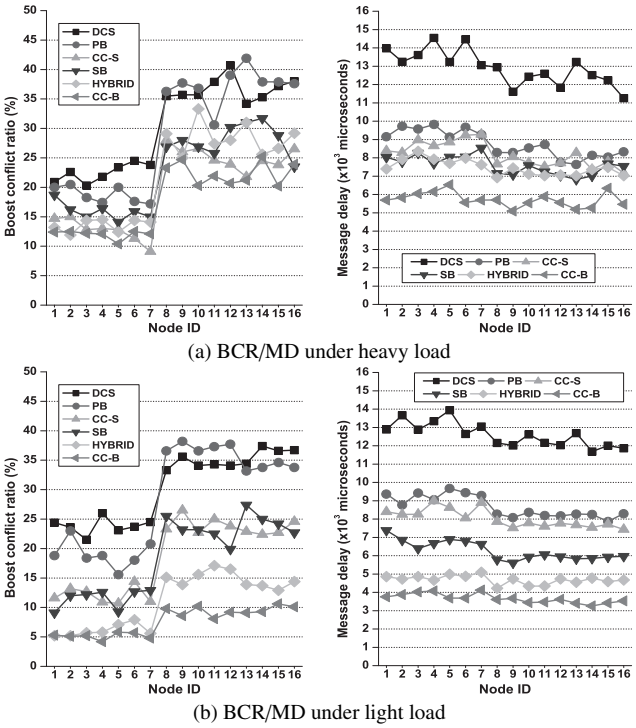
Table 3 Parallel workloads.

	Applications	Comm. Intensity	# of Proc.
WL1	BT	low	4, 9, 16
WL2	SP	medium	4, 9, 16
WL3	FT	medium	4, 8, 16
WL4	IS	high	4, 8, 16
WL5	CG	high	4, 8, 16
WL6	BT+SP+FT+IS+CG	low+medium+high	4, 8, 9, 16



(a) heavy load (b) light load

Fig. 7 Average job response time of all schemes for WL6 under heavy and light load.



(a) BCR/MD under heavy load

(b) BCR/MD under light load

Fig. 8 BCR/MD of all major schemes for WL6 under heavy and light load.

Thus, for the job mixes (WL6 case), we imitate the arrival pattern of jobs in a real system by randomly generating several jobs with different number of processors per job (The number of processors is selected from 4, 8, 9 or 16 as per the NAS application requirements.). These jobs arrive at the cluster with exponentially distributed inter-arrival times (Inter-arrival time distribution analysis of several supercomputer centers' workloads [16] showed an exponential fit.).

In our experiments, we set the maximum Multi-Programming Level (MPL) to five for the local scheduling and all CDC techniques. We limit the total size of simultaneously running applications to fit well into our memory (512 MB), and hence, we incur no swapping overheads. Considering swapping effects on coscheduling is an extensive research in itself, and is out of scope of our work in this study. We also use the default FIFO allocation policy provided by OpenPBS [13] in the front-end node. To analyze the performance of difference scheduling techniques, we consider the average job response time as the main performance metric, where the time is the sum of the waiting time and execution time.

5.3 Impact of Job Arrival Pattern for Mixed Workload

Figure 7 (a) and 7 (b) depict the average job response time of eight different scheduling techniques under heavy and light workloads (i.e. the average job inter-arrival time are 40.75 and 70.27 seconds), respectively. As a metadata of Fig. 7, Fig. 8 shows boost conflict ratio (BCR) and message delay (MD) of major CDC techniques. Here, BCR means the number of times that more than two processes have pending messages at the moment of context switch from the total number of context switches, and MD represents the time difference between when a process starts to wait for a message arrival and when the receive operation completes (i.e. the message is consumed by the process) averaged over all received messages. The results are obtained by running 100 mixed applications (WL6) from the NAS benchmarks. For this experiment, we limited the MPL to 5.

As expected, in Fig. 7 (a) heavy load, the average job response time of each scheduling scheme is much longer than that in Fig. 7 (b) light load. In Fig. 7 (a), the response times increase mainly due to large waiting times that jobs experience in the arrival queue, but in Fig 7(b), the difference between the execution times of the scheduling schemes is more pronounced. In Fig. 7 (a) and Fig. 7 (b), the results of the local scheduling (LOCAL) show the worst response times and DCS follows next. This is mainly because LOCAL makes no effort for coscheduling, resulting the longest waiting time. Of other six schemes, BATCH generally gives the next highest response time, due to its employment of space sharing only.

We also observe that the blocking-based schemes (SB, HYBRID, and CC-B) outperform the spinning-based schemes (LOCAL, DCS, PB, and CC-S) in all cases. Especially, from Fig. 8 (a) and Fig. 8 (b), it is apparent that the increment of system load does not affect the BCR and the MD in DCS and PB, which means that these schemes saturate in terms of utilization even under light load. In general, in spinning-based schemes, a communicating process spends most of its time spinning for a message, thus preventing

other ready processes from making progress. In blocking-based schemes, this wasted time is eliminated at a small cost of blocking and wake-up, providing more chances for other processes. This is the reason why the BCRs and MDs for DCS, PB, and CC-S are larger than those of SB, HYBRID, and CC-B, as shown in Fig. 8.

The most interesting observation from both figures is that CC-S and CC-B achieve significant performance improvements over other spinning and blocking-based alternatives, respectively. In addition, CC-B performs the best among all considered schemes. For example, CC-B reduces the average job response time by up to 36.6% (or 24.7%) compared to SB (or HYBRID), and CC-S by up to 8.02% compared to PB under heavy load. Although CC-S alters the scheduling order of conflicting processes, the time spent spinning considerably limits CC-S from attaining results that are competitive with those of CC-B.

From Fig. 8, another interesting observation is that CC experiences much lesser BCR (Boost Conflict Ratio) and MD (Message Delay) than existing CDC schemes (for example, CC-B reduces BCR and MD, by up to 22% and 25%, respectively, compared to SB under heavy load). Remind that CC dynamically adjusts the scheduling sequence of conflicting processes based on the rescheduling latencies of their correspondents in remote nodes. Thus, when resolving boost conflicts, delaying local processes communicating with their correspondents in the contended node (i.e., the node that has processes with pending messages competing for its CPU) helps the node to schedule other stringent processes without intervention, not introducing another excessive boost conflicts in that node. Also, scheduling in advance one of conflicting processes whose correspondents are in less contented nodes results in the remarkable decrease of MD within the system. From the reduction in both BCR and MD, we can find that the proposed CC makes beneficial decisions on boost conflicts, showing the effectiveness of ERL (Expected Rescheduling Latency) and W_r (normalized proportional weight based on ERL in order to establish an improved scheduling order in local scheduler) (see Eqs. (1)-(4) in Sect. 4.2).

In summary, the primary reason the CC approach outperforms others is that it rearranges the scheduling sequence of conflicting processes according to the rescheduling latency of their correspondents in remote nodes. According to these results, we can conclude that the proposed CC approach is very promising, especially for the hunting for wasted idle cycles in clusters, improving the overall system utilization, and this is in good agreement with the simulation results reported in [6].

5.4 Impact of the Nature of Parallel Applications

Next, we focus on the performance variation of different scheduling schemes when the communication intensities and patterns of the workloads change. As in Table 3, WL1 - WL5 are different types of workloads in terms of computation granularity, communication pattern, and message size distribution. In all 16 nodes, we simultaneously run 50 identical type of jobs, and we again fixed MPL to 5.

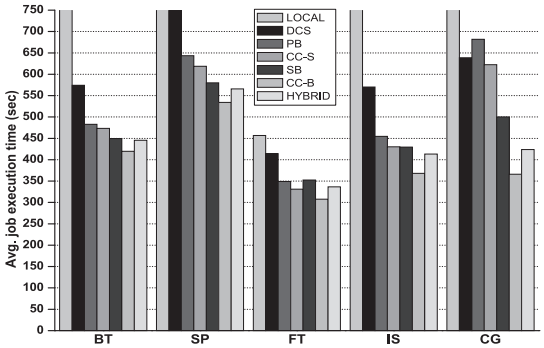


Fig. 9 Parallel application performance.

Figure 9 shows the average job execution times of seven scheduling schemes (excluding BATCH) for each workload WL1 - WL5. Note that the avg. job execution time measured in this experiment, is a system-perspective performance metric closely related to the system throughput. From the figure, we clearly confirm that with the use of reordering technique, CC-B and CC-S always outperform other blocking-based (SB and HYBRID) and spinning-based (LOCAL, DCS, and PB) alternatives, respectively. Of these, CC-B performs the best across all considered workloads, showing the shortest avg. job execution time.

In Fig. 9, for application with low communication intensity (like BT), the execution time difference between CC and other CDC schemes is less distinguished, although the proposed CC approach shows marginally better performance. As the communication intensity of workload increases (WL1 < WL2 < WL3 < WL4 < WL5), however, the performance benefit of the CC schemes over other schemes becomes more pronounced. For example, in WL5 (CG), CC-B shows much shorter execution time as high as 26.8% (or 13.6%) compared to SB (or HYBRID). From the results of this experiment, we also reconfirm that blocking-based schemes outperform spinning-based ones.

5.5 Impact of Multi-Programming Level (MPL)

Advancing CPU speed, memory size, and most importantly, reduced context switching overheads in an optimized OS kernel like Linux, there is enormous potential for improving system utilization through a high degree of multiprogramming. Thus, it is worthwhile to investigate how different scheduling techniques perform when MPL varies.

Figure 10 and Figure 11 show the changes of average job response time and BCR/MD of seven scheduling schemes for the mixed workload (WL6) when MPL value goes from 3 to 5, respectively. In this experiment, we exclude BATCH because it only supports space sharing. First of all, in Fig. 10, we observe that with increasing the MPL, the average job execution times of all schemes are uniformly prolonged. Of all schemes, LOCAL and DCS schemes have a very steep increase in execution time, whereas the other

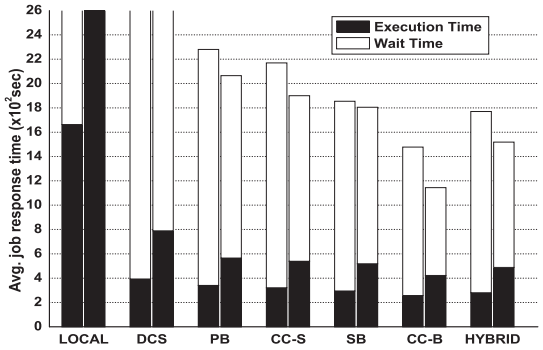
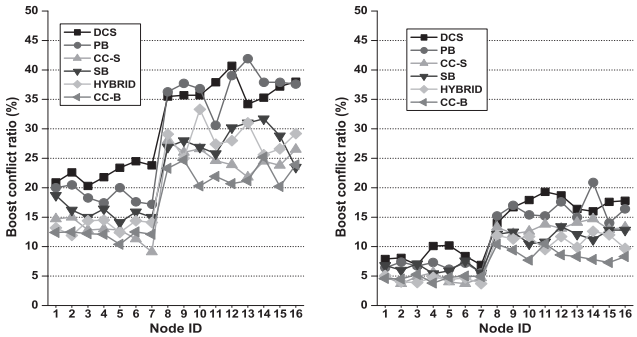
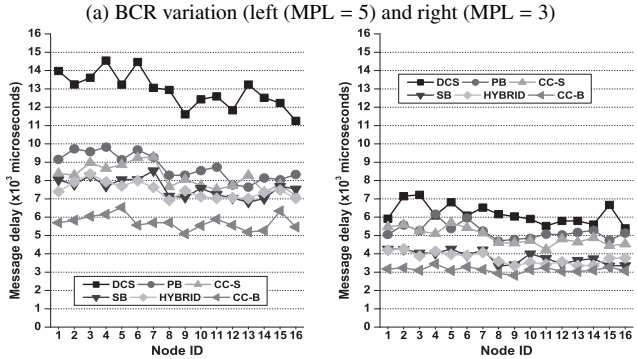


Fig. 10 Avg. job response times when MPL goes from 3(left) to 5(right).



(a) BCR variation (left (MPL = 5) and right (MPL = 3)



(b) MD variation (left (MPL = 5) and right (MPL = 3)

Fig. 11 BCR and MD variation between MPL = 5 and MPL = 3.

schemes are much more tolerant. In a larger MPL, however, due to additional free slots available in each node, we can allocate reasonably higher number of jobs; this lowers the waiting time per job. Overall, with the exception of LOCAL and DCS, other five schemes experience the decrease of the average job response time with increasing MPL. Of these, the CC schemes outperform other schemes at all MPLs.

At a lower MPL, the performance gain of the CC schemes is rather limited. In Fig. 10, the CC schemes reduce the response time by the small factor of up to 20.3% compared to the other schemes at MPL = 3. This is because each node has at most two competing processes, making the chance to apply our reordering function to become low, as shown in Fig. 11 (a). Larger MPL, however, permits a larger number of jobs to be simultaneously accommodated in the system, increasing the likelihood of a boost conflict at each context switch; this makes the scheduling sequence of conflicting process to be frequently affected by our reordering technique. Consequently, as we increase MPL, the proposed CC schemes provide better improvement in both the resulting response time and the message delay (see Fig. 11 (b)) over other CDC schemes.

6. Conclusions

In this paper, we explored the design issues and implementation details of novel contention-aware coscheduling (CC) strategy. CC dynamically resolves priority boost conflicts with detecting short-term load imbalance across nodes and regulating the scheduling sequences of conflicting processes based on it. With a generic coscheduling framework, we also performed broad spectrum of experiments on real multi-programmed heterogeneous clusters to analyze how different system parameters and job characteristics impact on the performance of CC and other CDC approaches.

From the results, we confirmed that i) priority boost conflict is common in multi-programmed clusters that deploy CDC mechanisms, and therefore should be carefully handled to improve system utilization and ii) significant performance improvement can be achieved with our CC schemes. The main reason for this improvement is that CC seeks to avoid unnecessarily wasted spinning and idle time by attaining a marked reduction in false decisions upon the boost conflict. The performance gain is even more pronounced when a communication-intensive workload and/or a larger MPL are applied to the system.

In the future, we will focus on evaluating the CC performance on large-scaled multi-core clusters as well as devising a revised model to more accurately estimate the contention on the resources in the system. We also plan to expand our work to the virtual machines (VMs) scheduling in cloud computing platform.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant number 20110026740) and also supported by the GRRC SUWON 2011-B5 program of Gyeonggi province.

References

[1] T.E. Anderson, D.E. Culler, and D.A. Patterson, "A case for NOW (networks of workstations)," IEEE Micro, vol.15, no.1, pp.54–64, 1995.

[2] F. Gine, F. Solsona, M. Hanzich, P. Hernandez, and E. Luque, "Cooperating coscheduling: A coscheduling proposal aimed at non-dedicated heterogeneous NOWs," J. Computer Science and Technology, vol.22, pp.695–710, Sept. 2007.

[3] C. Anglano, "A comparative evaluation of implicit coscheduling strategies for networks of workstations," Proc. Ninth IEEE International Symposium on High Performance Distributed Computing, pp.221–228, 2000.

[4] G.S. Choi, J.H. Kim, D. Ersoz, A. Yoo, and C.R. Das, "Coscheduling in clusters: Is it a viable alternative?," Proc. 2004 ACM/IEEE conference on Supercomputing, Nov. 2004.
[5] A. Dusseau, R. Arpaci, and D. Culler, "Effective distributed scheduling of parallel workloads," Proc. ACM SIGMETRCDC Conference, pp.25–36, May 1996.
[6] J.L. Yu, J.S. Kim, and S.R. Maeng, "A runtime resolution scheme for priority boost conflict in implicit coscheduling," J. Supercomputing, vol.40, no.1, pp.1–28, March 2007.
[7] E.G. Bradford, Measuring the Scheduler Overhead, Available from http://www-106.ibm.com/developerworks/linux/library/lrt9/
[8] P. Strazdins and J. Uhlmann, "A comparison of local and gang scheduling on a Beowulf cluster," Proc. IEEE Conference on Cluster Computing, pp.55–62, 2004.
[9] B. Lawson, E. Smirni, and D. Puiu, "Self-adapting backfilling scheduling for parallel systems," Proc. International Conference on Parallel Processing, pp.583–592, Aug. 2002.
[10] The Message Passing Interface (MPI) Standard. Avaialble from http://www-unix.mcs.anl.gov/mpi/
[11] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C.R. Das, "Alternatives to coscheduling a network of workstations," J. Parallel and Distributed Computing, vol.59, no.2, pp.302–327, Nov. 1999.
[12] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C.R. Das, "A closer look at coscheduling approaches for a network of workstations," Proc. ACM Symposium Parallel Algorithms and Architectures, pp.96–105, June 1999.
[13] OpenPBS. Available from http://www.openpbs.org.
[14] J. Ousterhout, "Scheduling techniques for concurrent systems," Proc. 3rd International Conference on Distributed Computing Systems, pp.22–30, 1982.
[15] P. Sobalvarro, S. Pakin, B. Weihl, and A.A. Chien, "Dynamic coscheduling on workstation clusters," Proc. International Parallel Processing Symposium, pp.231–256, March 1998.
[16] A.B. Yoo and M.A. Jette, "The characteristics of workload on ASCI Blue-Pacific at lawrence livermore national laboratory," Proc. CCGrid2001, pp.295–302, May 2001.
[17] Y. Zhang, H. Franke, J.E. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," IEEE Trans. Parallel Distrib. Syst., vol.14, no.3, pp.236–247, 2003.
[18] N.J. Borden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su, "Myrinet: A gigabit-per-second local area network," IEEE Micro, vol.15, no.1, pp.29–36, 1995.
[19] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke, "Impact of workload and system parameters on next generation cluster scheduling," IEEE Trans. Parallel Distrib. Syst., vol.12, no.9, pp.967–985, 2001.
[20] Myrinet. Inc. MPICH-GM software. Oct. 2003. Available from http://www.myrinet.com
[21] NAS division. The NAS parallel benchmarks. Available from http://www.nas.nasa.gov/Software/NPB

Jung-Lok Yu received his Ph.D. degrees from KAIST, Daejeon, Korea in 2007. He was a senior engineer in Samsung Electronics from 2007 to 2010. He is currently a senior researcher with Supercomputing center, Korea Institute of Science and Technology Information. His research interests include parallel processing, cluster computing, and cloud computing.

Hee-Jung Byun received the Ph.D. degrees from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2005. She was a senior engineer in Samsung Electronics, Ltd. from 2007 to 2010. She is currently an assistant professor with the Department of Information & Communications, Suwon University, Kyunggi-do, Korea. Her research interests include network modeling, controller design, and performance analysis.