J Supercomput (2007) 40: 1–28
DOI 10.1007/s11227-006-0006-3

A runtime resolution scheme for priority boost conflict in implicit coscheduling

Jung-Lok Yu · Jin-Soo Kim · Seung-Ryoul Maeng

Published online: 9 March 2007

Abstract High-performance parallel and scientific applications are composed of multiple processes running on distinct CPUs that communicate frequently. Due to the synchronization needs of such applications, performance is greatly hampered if their processes are not scheduled simultaneously on the CPUs. Implicit coscheduling (ICS) is a well-known technique to address this problem in multi-programmed clusters, however, traditional ICS schemes do not incorporate steps to adequately deal with priority boost conflicts, leading to significantly degraded performance. In this paper, we propose the use of runtime difference in *contention* across nodes to provide more sophisticated coscheduling decisions in response to the conflicts. We also present a novel coscheduling scheme termed PROC (Process ReOrdering-based Coscheduling) that adaptively regulates the scheduling sequence of conflicting processes based on the rescheduling latency of their correspondents in remote nodes. We perform extensive simulation-based experiments using both synthetic and realistic workloads to analyze the performance of PROC compared to alternatives such as local scheduling, a widely used batch scheduling, gang scheduling, and existing ICS schemes. The results show that all ICS schemes commonly experience priority boost conflicts, and that the proposed PROC significantly outperforms other ICS alternatives (or batch scheduling) by up to 50.4% (or 72.5%) in the average job response

J.-L. Yu (✉) · J.-S. Kim · S.-R. Maeng
Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, 305-701, South Korea
e-mail: jlyu@calab.kaist.ac.kr

J.-S. Kim
e-mail: jinsoo@cs.kaist.ac.kr

S.-R. Maeng
e-mail: maeng@calab.kaist.ac.kr

time. This improvement is achieved by reducing wasted idle time and spinning time without sacrificing fairness.

Keywords Clusters · Coscheduling · Priority boost conflict · Runtime contention · Rescheduling latency · Process reordering · Performance comparison

1 Introduction

Cluster systems [2] have recently gained increased acceptance as general-purpose multi-programmed computing servers for a variety of scientific and business applications [5, 31]. This trend makes the design of an efficient scheduler even more crucial and challenging. The ultimate goal of such schedulers is to effectively utilize underlying system resources. Time-sharing approaches are particularly attractive because they enable overlapped executions of processes with diverse demanding characteristics for shared resources [3, 23]. The simplest approach to time-sharing a cluster is to leave each node to schedule its own processes independently. Unfortunately, this form of scheduling can be very inefficient when running parallel jobs that need process synchronization, mainly due to the lack of coordination among local schedulers [3, 7, 19].

Over the years, two main strategies for coordinating individual local schedulers have been proposed: *gang scheduling* (GS) [21, 23] and *implicit coscheduling* (ICS) [1, 3, 5, 7, 19, 20, 27, 31]. 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 [8, 29]. It also does not scale well with the system size due to the global synchronization overhead [5, 31]. Recently, a full appreciation of these practical limitations of GS has led to a myriad of ICS schemes such as DCS [27], SB [7], PB [20, 31], and HYBRID [5]. These schemes use implicit 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, ICS schemes have better scalability and reliability, and have been shown to be efficient in clusters [5, 19].

The major problem with existing ICS schemes is that they overlook the importance of *priority boost conflicts*, failing to properly handle such conflicts.[1] A boost conflict is defined as a situation that involves two or more processes in a scheduling queue that have pending messages (waiting to be coscheduled with its correspondents in remote nodes) and a local scheduler has to schedule one of them at the next context switch. This generally occurs when multiple incoming messages are destined to distinct processes during a short period of time. To select a candidate among conflicting processes, existing ICS schemes mainly rely on local information such as message arrival, local process state; however, such an inattentive decision can increase the

[1]The terms, *priority boost conflict* and *boost conflict*, are used interchangeably in this paper.

likelihood of uncoordinated executions of parallel processes and lead to a significant degradation of system utilization.

We emphasize that understanding the *contention* of remote nodes is critical to achieve more beneficial decisions for the boost conflict. In an ICS context, a node contention represents the number of different stringent processes, e.g., processes with pending messages, which are disturbing the re-execution of the process that is currently running on the node. For example, let us assume that a process Q in a remote node sent a message to its correspondent P in a local node, and Q is spinning or blocks for a message from P. The process Q in the remote node has a greater likelihood of being rescheduled on new message arrivals (and thus, to be synchronized with P) if the remote node has little effectively stringent work. Therefore, as long as the remote node has low contention, scheduling process P first among all conflicting processes is strongly recommended. Otherwise, process P might miss a golden chance to be synchronized with its correspondent Q in the low-contended node. In addition, the node will suffer from wasted spinning and idle time. If each local scheduler has the capacity to discriminate the difference in contention among remote nodes, a new scheduling order of conflicting processes can be established so as to reduce wasted time in the system.

In this paper, we propose a flexible ICS scheme termed Process ReOrdering-based Coscheduling (PROC) that aims to efficiently resolve priority boost conflicts. Unlike existing ICS schemes, PROC 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 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 information notified from remote nodes, PROC grants 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. This approach can minimize wasted CPU time in the system as well as increase the opportunity of achieving synchronization among parallel processes, thus improving overall system utilization.

The remainder of this paper is organized as follows. In Sect. 2, we briefly review existing ICS schemes. Section 3 provides a detailed description of the proposed PROC scheme, while Sect. 4 explains the simulation methodology used to evaluate PROC. In Sect. 5, we analyze performance results that involve synthetic and realistic workloads. Finally, we conclude with a summary in Sect. 6.

2 Background and related work

In this section, we summarize native local scheduling, gang scheduling [21, 23], and all prior implicit coscheduling (ICS) [3, 19] techniques.

Local scheduling (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. While appealing in its simplicity, it yields unacceptable performance due to the lack of coordination among local schedulers.

Table 1 Implicit coscheduling (ICS) schemes

Type	Scheme	Msg. waiting action		Msg. handling action
		Sender	Receiver	(Boost Policy)
Spinning-based	LOCAL	Spin	Spin	Nothing
	DCS	Spin	Spin	Interrupt & Boost
	PB	Spin	Spin	Periodic Boost
Blocking-based	IB	Spin	Imme. Block	Interrupt & Boost
	SB	Spin	Spin-Block	Interrupt & Boost
	HYBRID	Imme. Block	Imme. Block	Boost & Restore (only collective comm.)

On the other hand, gang scheduling (GS) ensures that the processes of a job, called a *gang*, are scheduled and de-scheduled at the same time on different CPUs for a given time quantum. To achieve this, GS uses global knowledge constructed *a priori*, i.e., Ousterhout matrix [14, 21] and performs explicit global context switch. GS is an efficient coscheduling algorithm for fine-grained parallel jobs. However, it usually requires long time quanta to offset the high global synchronization overhead, making the system less responsive for interactive and I/O-intensive jobs. Moreover, GS can suffer from both external and internal fragmentation when jobs do not make use of the entire nodes (i.e., all slots within the same row of the matrix) and are not well-balanced in terms of communications, respectively [29, 31]. This situation leads to low CPU utilization.

As intermediate approaches between LOCAL and GS, ICS schemes can be classified by two components [5, 19]: *message waiting action* taken by processes waiting for a message and *message handling action* performed by the kernel when a message arrives (Table 1).

Dynamic CoScheduling (DCS) [27] uses an incoming message as a hint that the sender is currently scheduled on the remote node. In DCS, a receiving process performs spinning. Upon the arrival of a message, network interface card (NIC) checks whether the destination process of the message is the same as the process currently running on its host CPU. If there is a mismatch, an interrupt is raised. The interrupt service routine (ISR) boosts the priority of the destination process to coschedule it with the sending process. Periodic Boost (PB) [20] is an alternative scheme used to avoid expensive interrupt costs. In PB, the receiver is busy-waiting, as with DCS; however, rather than raising an interrupt for each incoming message, a kernel thread periodically checks message queues of each process in a round-robin way and boosts the priority of one of the processes with pending messages. There are several heuristics [31] used to decide which process to boost at each thread invocation. These heuristics only consider the local states of processes with pending messages in selecting a candidate for a priority boost.

Unlike the former spinning-based schemes, the following three schemes are blocking-based schemes that 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) [7] 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 [5], 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.

From the above description, it is apparent that existing ICS schemes do not devote any attention to determining a suitable scheduling sequence of conflicting processes for a priority boost conflict. In DCS, SB, and IB, priority boosts are statically given based on the order of message arrival, while in PB they are given in a simple round-robin manner. HYBRID gives precedence to processes joining in collective communication for the priority boost. Without understanding the conditions of other remote nodes, such a simple scheduling decision can significantly degrade system performance by generating wasted spinning and idle time within the system. To overcome these limitations of existing ICS schemes, in the next section we introduce a coscheduling scheme that regulates the scheduling sequence of conflicting processes based on the rescheduling latency of their correspondents.

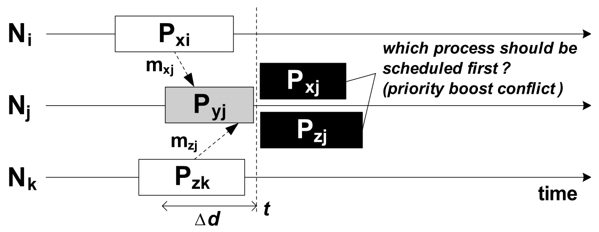
3 Process ReOrdering-based Coscheduling (PROC)

3.1 Motivation and basic idea

This work is motivated from the observation that undertaking a coscheduling decision in each local scheduler can be significantly complicated by load imbalance. Load imbalance commonly arises in multi-programmed clusters for the following reasons [11, 24]:

1. Application imbalance: uneven load (computation, I/O, and communication) distribution to equally powerful computing nodes. Each job requires a different number of nodes with different computation granularity, communication patterns, and message size.
2. Heterogeneity in cluster hardware resources: e.g., different CPU speed, memory hierarchy, or even different number of CPUs per node.
3. Multi-programming: the presence of local users and background jobs that interfere with the execution of parallel jobs.

Even if all the parallel jobs are well-balanced with regular communication patterns, they still reveal different scheduling characteristics due to the load imbalance caused by points (2) and (3) above, thereby imposing irregular communication traffic on

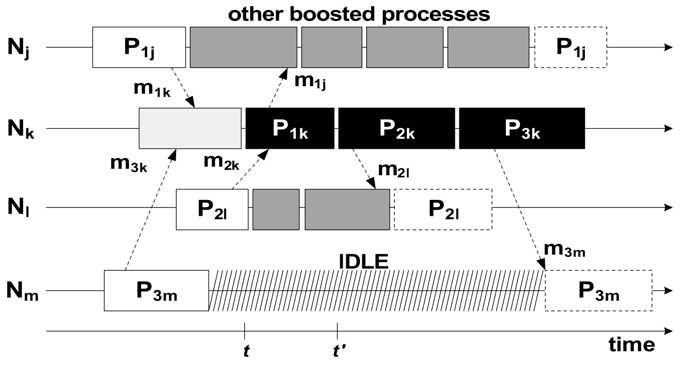

N_i
N_j
N_k
P_xi
P_yj
P_zk
P_xj
P_zj
m_xj
m_zj
which process should be scheduled first ? (priority boost conflict)
time
t
Δd
Fig. 1 Example of a priority boost conflict

the network. 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., as indicated in point (1) above.

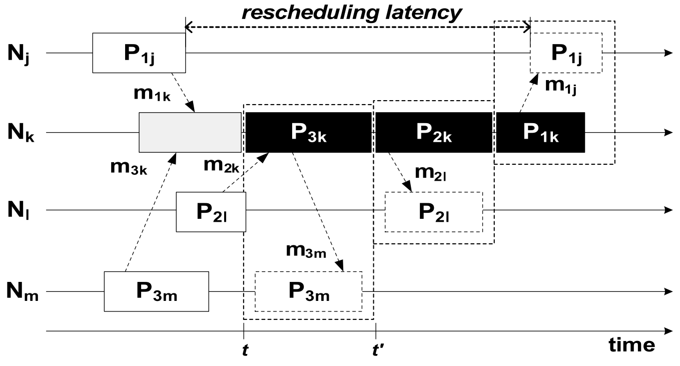
Priority boost conflicts normally occur on these multi-programmed clusters. Figure 1 depicts the occurrence of a boost conflict. Let P_{ij} denote the process of a parallel job i running on node N_j, while P_{i*} represents all the processes that belong to the same job i. m_{ij} is a message destined 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. The boost conflict's probability of occurring increases as the degree of multi-programming and/or the number of communication-intensive jobs increase.

Scheduling conflicting processes should be carefully determined to optimize system utilization. Nevertheless, as mentioned in Sect. 2, existing ICS schemes tend to make a simple decision when faced with a boost conflict. To illustrate this point, Fig. 2a shows an example of a scheduling sequence of conflicting processes in existing ICS 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 ICS 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. In addition, the priority boost given to P_{1j} causes additional excessive contention on the CPU, and thereby potentially prevents other boosted processes in N_j from being coordinated with their correspondents.



(a) existing ICS schemes



(b) PROC

Fig. 2 An example scheduling sequence for conflicting processes

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 (or unnecessary spinning) state. This leads to low resource utilization.

The basic idea of PROC 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 2b provides a conceptual depiction of PROC. Note that, as in Fig. 2a, N_k has conflicting processes at time t and t'. As shown in Fig. 2b, 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, PROC can not 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.

To put this idea into practice, two key questions must be answered: (1) how do we measure and propagate the rescheduling latency information of each node with low overhead and (2) how do we exploit and manage the information to arrive at more accurate coscheduling decisions regarding the boost conflict. In the next two subsections, we investigate these questions in detail.

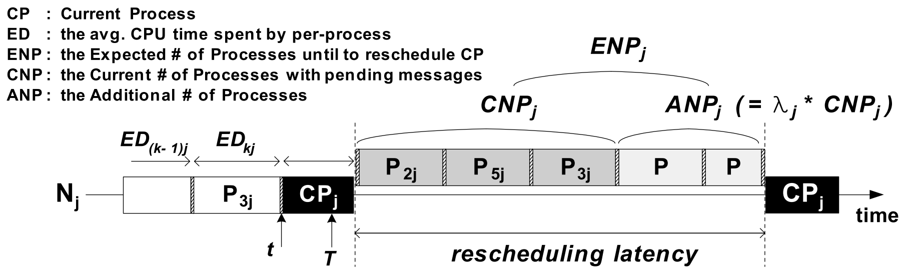
3.2 Estimating rescheduling latency

PROC estimates rescheduling latency at runtime to attain better decisions when resolving conflicting processes. The question is then how to formally define and measure the rescheduling latency. The effectiveness of the PROC approach is strongly dependent on the accuracy of the measured rescheduling latency information. In practice, however, it is too difficult to precisely measure the rescheduling latency and their discrepancy across nodes without incurring significant overhead. To minimize the overhead in a non-intrusive manner, the following heuristics are used.

We formally define the *rescheduling latency* of a node as the expected remaining time until the current process is scheduled again on the node. The basic mechanism used within ICS is to boost the priority of processes with pending messages, which is critical to the contention introduced on the shared CPU resource. Hence, in an ICS context, the rescheduling latency of a node N_j at time T, $resched\_lat(N_j, T)$ is mainly affected by the number of processes with unconsumed messages. Furthermore, the rescheduling latency also increases as the ratio of new processes whose state changes to "ready" increases by new message arrivals and/or I/O completion. As the rescheduling latency is on the time-domain, each node measures the following two values:

- **E**xecution **D**uration (**ED**) : the estimated average CPU time spent by each process between consecutive context switches
- **E**xpected **N**umber of **P**rocesses (**ENP**) : the expected number of processes to be run on the node before the current process can be rescheduled

Figure 3 shows the measurement of the rescheduling latency information in PROC. Suppose that a node N_j has a current process (CP_j) that uses its CPU at time T. At each context switch, e.g., time t in the figure, N_j can calculate the average CPU time spent per process based on the last and past execution durations, as the following equation; $ED_j(k) = w \cdot ED_{kj} + (1-w) \cdot ED_j(k-1)$, where k is the number of context



CP : Current Process
ED : the avg. CPU time spent by per-process
ENP : the Expected # of Processes until to reschedule CP
CNP : the Current # of Processes with pending messages
ANP : the Additional # of Processes
ENP_j
CNP_j
ANP_j (= λ_j * CNP_j)
$ED_{(k-1)j}$
ED_{kj}
N_j
P_{3j}
CP_j
P_{2j}
P_{5j}
P_{3j}
P
P
CP_j
time
t
T
rescheduling latency

Fig. 3 Estimation of the *rescheduling latency* information in each node

switches before time t, ED_{kj} is the last execution duration known at time t, w ($0 \leq w \leq 1$) is a weight factor, and $ED_j(0) = 0$. We have empirically found that $w = 0.2$ is enough good to detect well both the execution durations of different processes and their rapid changes. ED_j denotes the most up-to-dated $ED_j(k)$ value in N_j.

In Fig. 3, ENP represents the expected number of processes that have potential for being executed on the CPU when the current process is scheduled-out, and ENP_j is the most recently measured ENP value. ENP_j is calculated by summing both CNP_j (Current Number of Processes) and ANP_j (Additional Number of Processes). CNP_j is the number of processes with pending messages in N_j at time t; it corresponds to stringent processes to be executed immediately on the node for the coscheduled executions with their correspondents. ANP_j corresponds to invisible processes at time t that can be additionally generated while executing processes during the time interval $ED_j \times CNP_j$. In PROC, ANP_j is approximated by multiplying CNP_j by the rate λ_j at which new processes are placed in the highest-level ready queue (in the case of the Solaris scheduler [28]) by I/O completion and new message arrival during the time period ED_j. Consequently, the rescheduling latency of the node N_j at time T is obtained from the following equation:

$$resched\_lat(N_j, T) = (ED_j \cdot ENP_j) + \big((\lceil ENP_j \rceil + 1) \cdot switching\_cost\big).$$

In general, two values (ED_j and ENP_j) are updated at each context switch and are piggybacked in every outgoing message as the rescheduling latency information of the node. We assume that the cost of calculating ED_j and λ_j is negligible, as just a few arithmetic and bit operations are required for those operations. The calculation of ENP_j can induce a slight overhead because it is necessary to check whether a process has pending messages. This cost is modeled in the simulator.

3.3 Process reordering

The basic rationale of PROC is that whenever a boost conflict is detected, the process whose corresponding processes will be scheduled sooner in remote nodes should be scheduled first. To accomplish this, we need to first upload and manage the piggybacked rescheduling latency information, and secondly find a candidate process for the next context switch based on the available information.

As shown in Fig. 4, whenever a message arrives, the rescheduling latency information (ED and ENP) extracted from the message and a time-stamp (ts) are uploaded

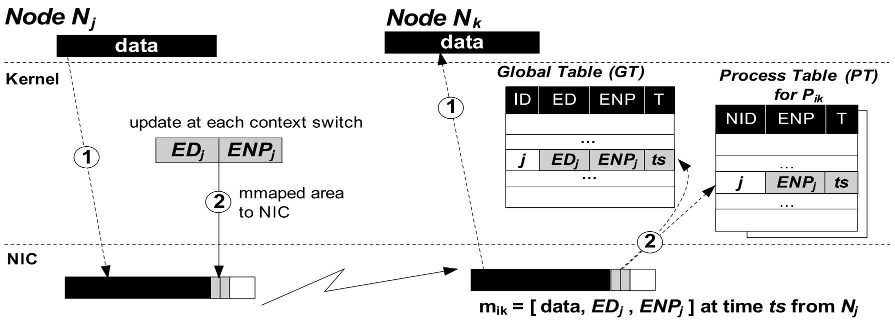

Node N_j
data
Node N_k
data
Kernel
update at each context switch
ED_j | ENP_j
mmaped area to NIC
Global Table (GT)
ID | ED | ENP | T
j | ED_j | ENP_j | ts
Process Table (PT) for P_{ik}
NID | ENP | T
j | ENP_j | ts
1
2
NIC
m_{ik} = [data, ED_j, ENP_j] at time ts from N_j

Fig. 4 Rescheduling latency information management

onto specific entries of two types of tables that can be accessed by the scheduling layer: the Global Table (GT) and the Process Table (PT). For each remote node, GT keeps the rescheduling latency information piggybacked with the last message sent by the node. For each local process, PT keeps the rescheduling latency information of its correspondents in remote nodes. In the case of a boost conflict, a candidate process can be selected based on the minimum rescheduling latency in PTs. The rescheduling latencies in PTs, however, are exposed to some staleness over time. Thus, we use more up-to-dated information in GT to approximately adjust entries in the PTs.

Algorithm 1 PROC algorithm

```
 1.  Input : $t$ (current time), Output: next_pid (process ID to run next)
 2.  function PROCESS_REORDERING
 3.  next_pid := NONE; mERL := infinite;
 4.  for each $p \in$ set of processes with pending message(s) do
 5.      i := $p$.id;
         /* for each message */
 6.      while $msg \neq$ NULL do
 7.          s := $msg.source$;
             /* maintain the obtained rescheduling latency information */
 8.          if (PT[i][s].T < GT[s].T)
 9.              PT[i][s].ENP := PT[i][s].ENP - ((GT[s].T - PT[i][s].T) / GT[s].ED);
10.              if (PT[i][s].ENP > GT[s].ENP)
11.                  PT[i][s].ENP := GT[s].ENP;
12.              end if
13.              PT[i][s].T := GT[s].T;
14.          end if
             /* find a local process with a minimum ERL value */
15.          ERL := (GT[s].ED * PT[i][s].ENP) - ($t$ - PT[i][s].T);
16.          if (ERL < mERL)
17.              mERL := ERL; next_pid := i;
18.          end if
19.          $msg$ := $msg$.next;
20.      end while
21.  end for
22.  return next_pid;
```

As each process with pending messages contains a list of the rescheduling latency information of its correspondents, our process reordering function can establish a new improved scheduling order among conflicting processes. Algorithm 1 shows the pseudo code of the reordering function.

For any given message (*msg*), the Expected Rescheduling Latency (ERL) value represents the expected remaining time until its corresponding process in a remote node (s) is rescheduled. From lines 6 to 20, we see that one of the conflicting processes whose correspondent has a minimum ERL is chosen. From lines 8 to 14, for each *msg*, the function updates the related rescheduling latency information in PT. As processes receiving messages from the same remote node (s) share the most recent state of the node, PT entries are updated using the new and existing information in GT and PT. The aim of these lines is to adjust the decline speed of the ERL according to the change of the ED and ENP values of the same remote node. From lines 15 to 19, the ERL of a given message is computed by extracting the elapsed time from the total expected rescheduling latency; the candidate process (next_pid) with the minimum Expected Rescheduling Latency (mERL) is thereby obtained. Consequently, this reordering function enforces the native scheduler to adaptively select the most urgent process that communicates with the correspondents with the shortest rescheduling latency.

Note that PROC is a complementary approach to existing ICS algorithms in that it only reorders the scheduling sequence of processes upon boost conflict. Therefore, PROC can be easily integrated with existing schemes. We present two schemes as examples to verify the effectiveness of the proposed approach: PROC-S (PROC-Spinning) and PROC-B (PROC-Blocking). In PROC-S, processes are busy-waiting both at senders and at receivers, and a periodically invoked kernel thread (similar to PB) performs process reordering based on the minimal rescheduling latency. In contrast, PROC-B uses immediate-blocking at both sides if communications are not completed. In PROC-B, our reordering function is applied at each context switch.

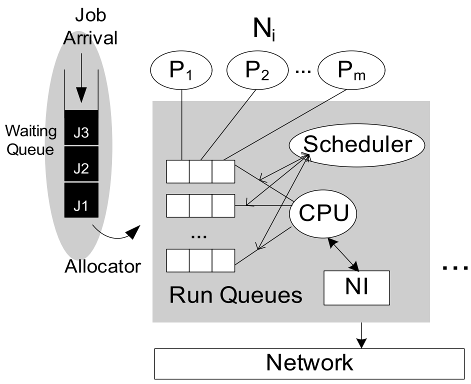
4 Experimental environments

Here we evaluate the performance of the proposed PROC schemes and compare it with those of a broad spectrum of scheduling alternatives, including a widely used batch scheduling (BATCH), GS (based on the implementation of GangLL [14] scheduler), LOCAL, and four representative ICS schemes (DCS, PB, SB, and HYBRID). Before we present the detailed discussion of the results, we describe the simulation platform, workloads, and the performance metrics used in this study.

4.1 Simulator

We implemented the proposed PROC schemes onto an extensible/unified simulation framework termed ClusterSchedSim [30] built on CSIM19 [26] toolkit. Figure 5a depicts the overall system model, which consists of an allocator and a set of nodes connected by a fast network. An arriving job is allocated the required number of nodes if available. Otherwise, the job is blocked in a waiting queue until there are enough free nodes. The network module connects nodes. As this work mainly focuses on scheduling, we use a simple linear model for the network that is parameterized by the message size.

Each node comprises a network interface (NI), OS scheduler, and the user process. On a message arrival from the network, NI delivers it to a user buffer and raises an interrupt if required. Similarly, NI waits for outgoing messages and enqueues them into the network. This is a typical form of operation in user-level communications [6, 15]. Costs for these operations have been obtained from a microbenchmark on a cluster of Pentium-4 1.8 GHz nodes connected by Myrinet [4]. The Scheduler module emulates the Solaris scheduler [28] and manipulates a priority-based multi-level feedback queue (60 queues) on which ready-to-run processes are placed. Each node can handle a Multi-Programming Level (MPL) of user processes, whose executions are expressed by a simple language that allows the specification of computations, I/O,



(a) simulation model of cluster system

Parameters		Value(s)
System size (# of nodes)		32, 64
Multi-Programming Level		2, 3, 5
Communication patterns		NN, AA
Message size		16 KB
One-way message latency		229.9 µs
Task skewness (v)		0%, 10%, 60%
Context switching	ICS	100 µs
	GS	1 ms
Gang scheduling quantum		0.1 s
Interrupt processing cost		50 µs
Check a process whether it has pending messages		2 µs
Upload/download of rescheduling latency info.		2 µs/1 µs
Update&compare a PT entry		0.5 µs
Move the position of a process		3 µs

(b) simulation parameters and values

Fig. 5 Description of the simulation environment

and communications. We assume that the total memory requirement of user processes in each node does not exceed the physical memory size of the node.

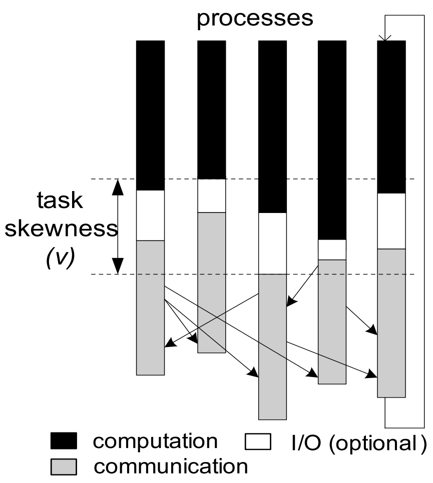
For supporting ICS schemes, each node has three additional modules: interrupt service, periodic boost, and reordering. The interrupt service module is used in DCS, SB, HYBRID, and PROC-B. It is invoked immediately after NI raises an interrupt. After a certain amount of interrupt processing time, the scheduling queues are manipulated to wake up (or to boost the priority of) the corresponding process. The periodic boost module is used in PB and becomes active every one millisecond, as described in [31], to boost the priority of the process with pending messages in a round-robin manner. The reordering module implements our reordering function. It is activated every millisecond in PROC-S, or at each context switch in PROC-B.

For SB, we set the *spin time* for a message to be the expected one-way latency. For HYBRID, we differentiate between the computation and the collective communication (e.g., all-to-all, barrier) phases in a parallel process by setting/unsetting a flag. When any process with the flag set is available, we give precedence to the process at each context switch. In both PROC-S and PROC-B, costs for calculating ENP, downloading/uploading the rescheduling latency information to the NI/scheduling layer, updating and comparing the ERL value in each PT entry, and changing the position of a process in the scheduling queue are all modeled in the simulator. The key simulation parameters used in the experiments are summarized in Fig. 5b.

4.2 Workloads and metrics

We consider two types of workloads: synthetic and realistic. Synthetic workloads are generated from SDSC IBM SP2 traces [22], which are widely used in many scheduling studies [9, 25]. During workload generation, job arrival time, execution time, and size information are characterized to fit the accurate workload model in [12]. A job can have different proportions in three components: computation, communication,

Job type			
	Comp.	Comm.	I/O
J1	85 (%)	15	–
J2	50	50	–
J3	35	65	–
J4	30	30	40
Workloads (with NN and AA)			
*wl*1	a set of J1		
*wl*2	a set of J2		
*wl*3	a set of J3		
*wl*4	equal mix of J1, J2, J3		
*wl*5	equal mix of J1, J3, J4		



(b) parallel job execution (a) synthetic workload characteristics

Fig. 6 Workload characteristics and parallel job execution

and I/O. For the evaluations, we identify four different job types with different proportions of these components, while five different workloads, termed $wl1 \sim wl5$, are used, as shown in Fig. 6a.

Each job requires a set of nodes ranging from 2 to 32, where each process iterates a loop with local computation, I/O, and inter-process communication (see Fig. 6b). We consider two communication patterns: Nearest Neighbor (NN) and All-to-All (AA), as commonly used in many parallel applications. We assume that both patterns use a fixed message size of 16 KB. By fixing the end-to-end one-way latency of a message, the communication time per loop can be calculated under the assumption that processes are perfectly balanced. Using this time and the proportions of the other two components, we calculate the computation and I/O times per loop. In addition, we model the task skew by multiplying the computation and I/O times by a value uniformly selected in $(1 + \mathit{unif}(-v, v))$ and by varying the skewness factor (v).

Our workloads are then completed by five realistic parallel applications, BT, MG, FT, IS, and CG, which have been directly derived from the NAS Parallel Benchmarks (NPB) suite [18]. The choice of above applications is based on their different computation granularity, communication intensity, and pattern. In particular, the applications have been obtained by translating their source codes in NPB into the language accepted by the simulator, without changing their execution flow, communication topology, and message sizes. We used 64 nodes configuration and default input files from rather big problem size, CLASS = B. The duration of sequential parts of each application is determined from a run of the corresponding NPB application on our cluster. The characteristics of the five applications are listed in Table 2.

We use several metrics to compare the performances of different schemes:

Table 2 Realistic workload characteristics (64 nodes, CLASS = B)

	Comm. intensity (pattern)	# of msgs. /process	Msg. size distribution: bytes(%)
BT	Low (NN)	19200	6760(43.8%), 40560(43.8%), 90480∼91520(12.4%)
MG	Medium (NN,AA)	5702	8∼800(31.6%), 2048∼9248(11.7%), 32768∼69696(56.7%)
FT	Medium (AA)	10080	24(50%), 49152(50%)
IS	High (AA)	7560	4(33.3%), 4116(33.3%), 32768(33.3%)
CG	High (NN)	37950	8(59.3%), 16(1.2%), 75000(39.5%)

- *Average Job Response Time*: the time difference between when a job completes and when it arrives in the system averaged over all jobs.
- *Wait Time*: the average time spent by a job waiting in the arrival queue before it is scheduled.
- *Execution Time*: the difference between *Response* and *Wait* times.
- *Turnaround Time*: the total running time of the entire workload. It describes the reciprocal of the system throughput.
- *Utilization*: the percentage of time that the system actually spends in useful work.
- *Slowdown*: the ratio of the *execution time* to the time taken on a system dedicated solely to this job.
- *Fairness*: the fairness to different job types (CPU, communication, or I/O intensive) is evaluated by comparing the *slowdown* of the individual job types and its coefficient of variation in a mixed workload. A smaller variation indicates a fairer scheme.
- *Boost Conflict Ratio* (BCR): 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 (averages over all nodes).
- *False Decision Ratio* (FDR): the number of false decisions from the total number of boost conflicts (averaged over all nodes). In this paper, a false decision represents a situation where a newly scheduled process does not communicate with someone in the lowest contended node.[2]

To better understand the results, we provide a *Performance Profile*, which shows the percentage of time spent by a CPU on different components; computation, spinning, idling, context switches, and other overheads such as blocking, interrupt-processing, reordering costs, etc. "Idling" is further classified into the idle and the real-idle, which represents the idle time caused by inter-process communication dependency and that caused by the absence of jobs executed, respectively.

[2]The problem of maintaining current information regarding which node has the lowest contention is resolved by using Oracle, which has the latest rescheduling latency information of each node.

5 Performance results

5.1 Performance comparisons of all scheduling schemes

Figure 7 and Fig. 8 compare the average job response times of nine different scheduling schemes for the mixed workload (*wl*4) under light and heavy load,[3] respectively. For this experiment, we limited Multi-Programming Level (MPL) to 5 and fixed the task skewness factor (v) to 10%.

As expected, the heavy-loaded system provides much longer average job response time than the light-loaded system for all schemes. This is mainly due to the large waiting times that jobs experience in the arrival queue (see Fig. 7a and Fig. 8a). From Fig. 7b and Fig. 8b, it is apparent that the increment of system load does not affect the percentage of time spent in computation in LOCAL, DCS, and BATCH. This means that these schemes saturate in terms of utilization, even under moderately light load. GS and BATCH suffer from external resource fragmentation as inferred from the portion of real-idle component. This is because BATCH employs space sharing only and GS requires free slots on the required number of CPUs within the same row of the Ousterhout matrix. As a result, such restrictions enforce higher job waiting times in GS and BATCH. Note that the fragmentation can be mitigated by other allocation techniques, for example, backfilling [16].

We also observe that the blocking-based schemes (SB, HYBRID, and PROC-B) outperform the spinning-based schemes (LOCAL, DCS, PB, and PROC-S) in all cases. 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.

The most striking observation from Fig. 7 and Fig. 8 is that PROC-S and PROC-B achieve significant performance improvements over other spinning and blocking-based alternatives, respectively. PROC schemes achieve more pronounced results as

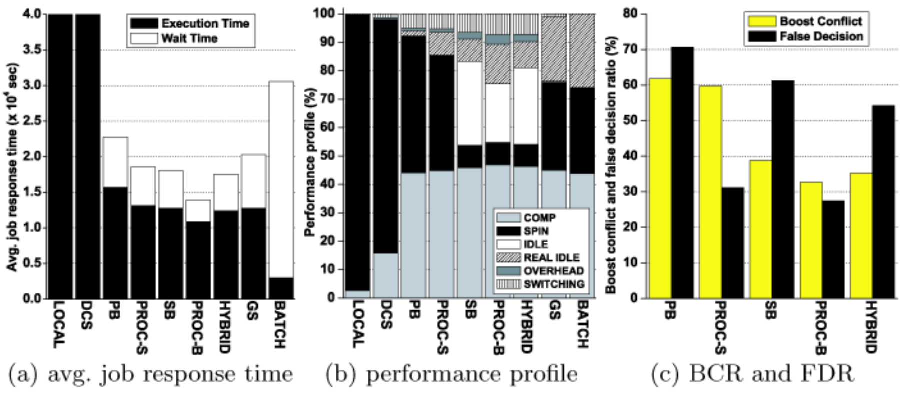


Fig. 7 Average job response times under light load (*wl*4, MPL = 5, $v = 10\%$)

[3]For a light load, the average job inter-arrival time is about 2.48×10^3 sec, and for a heavy load, it is about 1.55×10^3 sec.

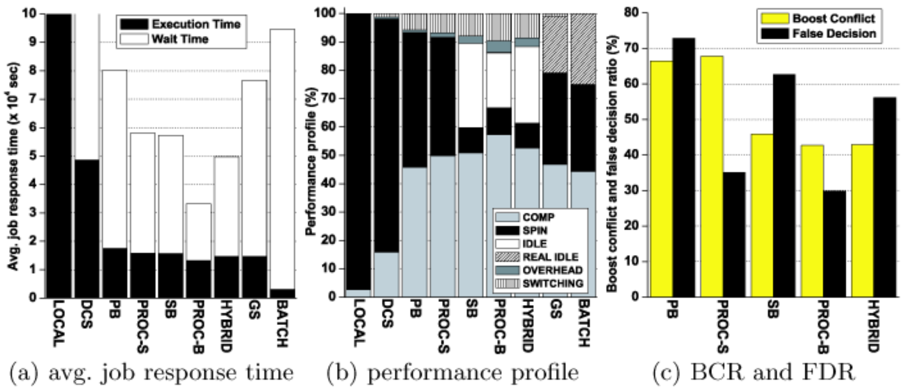


Fig. 8 Average job response times under heavy load ($wl4$, MPL = 5, $v = 10\%$)

the system load increases. In addition, PROC-B performs the best among all considered schemes. For example, PROC-B reduces the average job response time by up to 42.3% (or 33.2%) compared to SB (or HYBRID), and PROC-S by up to 27.7% compared to PB under heavy load. Although PROC-S alters the scheduling order of conflicting processes, the time spent spinning considerably limits PROC-S from attaining results that are competitive with those of PROC-B.

To understand why reordering-based schemes outperform other alternatives, we measured BCR and FDR of spinning-based schemes (PB and PROC-S) and blocking-based schemes (SB, HYBRID, and PROC-B), as shown in Fig. 7c and Fig. 8c. Even though all schemes show common results concerning BCR, PROC schemes get markedly low FDR values compared to existing ICS schemes. We note that the BCR values in spinning-based schemes are much higher than those in blocking-based ones. This is because a process's spinning in spinning-based schemes results in other processes slowly consuming already pending (or newly arriving) messages. We also find that the FDR values in HYBRID are slightly lower than those in SB due to the former's use of immediate-blocking and explicit priority boosting at collective communication phases. In contrast to other ICS schemes, PROC schemes rearrange the scheduling sequence of conflicting processes according to the rescheduling latency of their correspondents in remote nodes; therefore, the likelihood of false decisions is substantially reduced. PROC-B (or PROC-S) achieves a 52.5% lower FDR compared to SB (or 51.8% compared to PB). As a result, the beneficial decisions for the boost conflict pave the way for a significant decrement in unnecessary idle and spinning time (up to 44.6% under heavy load), and improve the overall system throughput.

PROC schemes incur some additional overhead. This mainly arises from the system times needed to update the rescheduling latency information in PTs for more accurate coscheduling decisions; however, these addition costs do not outweigh the achieved performance benefit. In the rest of this section, we set the inter-arrival time of jobs to be 1554.42 seconds with a mean execution time 3228.05 seconds unless otherwise mentioned.

5.2 Effect of communication intensity

Here, we investigate variations in the performance of different schemes under different communication intensities of workloads. As indicated in Fig. 6a, *wl*1, *wl*2, and *wl*3 represent CPU-intensive, evenly-balanced, and communicat-ion-intensive workload, respectively. Figure 9 depicts the average job response times and corresponding performance profiles for these workloads. Table 3 shows BCR and FDR of five ICS schemes for each workload. For this experiment, we again fixed MPL to 5 and v to 10%.

Table 3 BCR and FDR for five ICS schemes for *wl*1, *wl*2, and *wl*3 (%)

	*wl*1		*wl*2		*wl*3	
	BCR	FDR	BCR	FDR	BCR	FDR
PB	34.65	72.07	70.41	71.32	75.82	72.78
PROC-S	34.43	35.18	71.65	35.05	78.91	36.21
SB	29.26	57.55	49.14	62.94	52.79	62.70
PROC-B	26.71	19.11	49.42	31.89	50.03	33.50
HYBRID	29.19	56.63	46.63	55.42	48.58	52.57

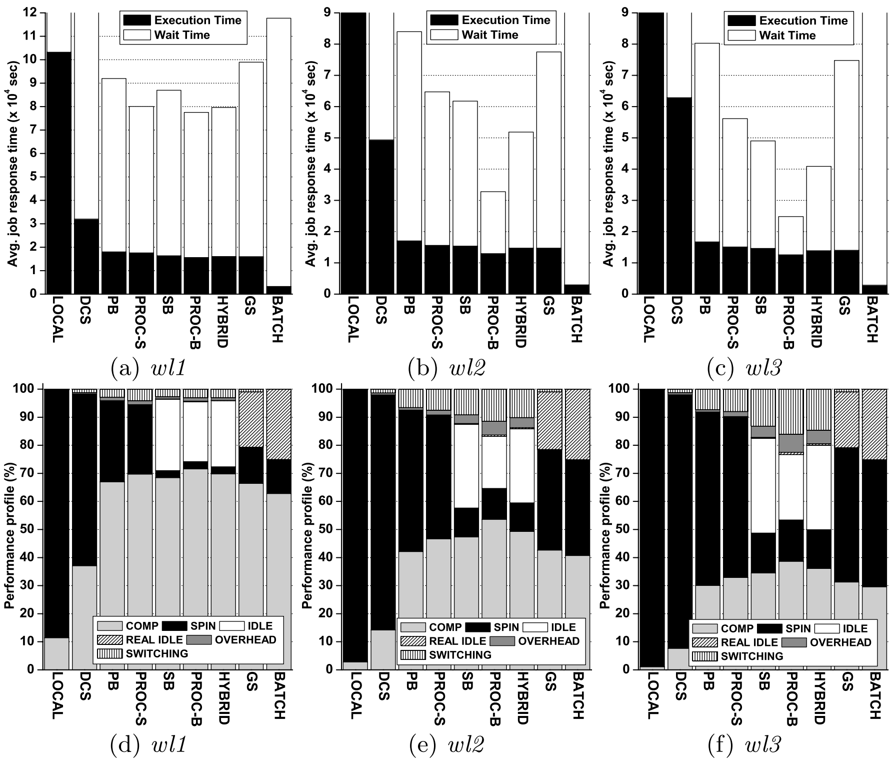


Fig. 9 Impact of communication intensity (MPL = 5, v = 10%)

In Fig. 9, LOCAL, DCS, and BATCH show poor response times compared to the other schemes, mainly because of a longer waiting time. These inefficiencies are amplified with increasing communication intensity. In general, GS has a higher response time compared to the other five ICS schemes (PB, PROC-S, SB, HYBRID, and PROC-B) in all three workloads. This is primarily due to the detrimental effect of task skewness (even a $v = 10\%$) and the restriction of resource allocation (i.e., allocating CPUs to a job in GS requires those CPUs to be available during the same time quantum). The skewness between the executing tasks of a job, however, gets lower with a lower computation fraction (from $wl1$ to $wl3$), making the inefficiencies of GS less important.

For the CPU-intensive workload ($wl1$), there is no significant difference in the resulting response times among PB, PROC-S, SB, HYBRID, and PROC-B, even if the PROC schemes still achieve marginally better performance. Due to infrequent communications in $wl1$, five schemes have low BCR values compared to those for other two workloads, $wl2$ and $wl3$, as shown in the first column in Table 3. Less messages being exchanged among processes also imply a smaller number of context switches due to infrequent priority boost. Therefore, when the computation portion becomes large, the detrimental effect of false decisions is well-hidden.

On the other hand, when the communication intensity of jobs increases, there is a markedly different outcome (see Fig. 9b, c, e, and f). The effectiveness of PROC schemes in terms of response times is increasingly pronounced; this result emphasizes the urgent need for addressing false decisions on the boost conflict. For instance, from Fig. 9c and Fig. 9f, we see that PROC-B substantially reduces the average job response time by 50.4% compared to SB, and by 72.5% compared to BATCH. A higher communication intensity of jobs causes more messages to be exchanged per unit time during executions, which increases the average number of processes waiting for a priority boost in each node. Consequently, this increases the number of both context switches and boost conflicts. This behavior can be easily gleaned by the time spent in switching component in Fig. 9f and by the BCR values in the second (or third) column in Table 3. In addition, a greater number of incoming messages means that a greater number of processes join in a boost conflict and their higher variation exists among nodes. Thus, the penalty of false decisions becomes higher for a communication-intensive workload than that for a computation-intensive workload, as low-contended nodes are likely to remain in their idle or spinning state for a longer time. Overall, existing ICS schemes, which determine the scheduling order of conflicting processes blindly, suffer even more from wasted time for $wl3$, as shown in Fig. 9f; however, the PROC schemes still attain more accurate scheduling decisions than existing schemes (see the FDR values in Table 3).

5.3 Effect of Multi-Programming Level (MPL)

Multi-Programming Level (MPL) is an important factor that has impacts on the system utilization and responsiveness. Hence, it is worthwhile to study how different scheduling schemes perform under varying MPL. In this experiment, we only consider six schemes (PB, PROC-S, SB, PROC-B, HYBRID, and GS) because LOCAL and DCS always show poorer performance than PB, and BATCH only supports space sharing (i.e., MPL is always one).

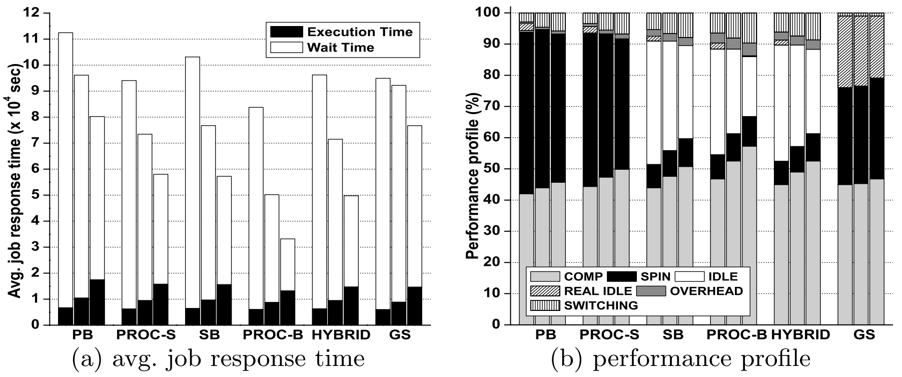


Fig. 10 Impact of MPL (*wl*4, MPL = 2(left-hand bars), 3(middle bars), and 5(right-hand bars); $v = 10\%$)

Table 4 BCR and FDR for five ICS schemes for MPL = 2, 3, and 5 (%)

	MPL = 2		MPL = 3		MPL = 5	
	BCR	FDR	BCR	FDR	BCR	FDR
PB	13.13	45.51	37.26	57.61	66.44	72.89
PROC-S	10.47	16.51	31.60	22.93	67.83	35.06
SB	15.39	55.26	29.64	58.25	45.90	62.71
PROC-B	14.69	16.93	29.15	23.12	42.73	29.81
HYBRID	15.43	49.49	28.17	50.85	42.95	56.19

Figure 10 shows the performance of six scheduling schemes for *wl*4 with MPL values from 2 to 5, while Table 4 shows BCR and FDR of each ICS scheme for the corresponding MPL level. First, it is apparent from Fig. 10a that an increase in MPL leads to increasing numbers of jobs entering into competition; thus the average job execution times of all schemes are uniformly stretched. However, because of additional slots available in each node, a lager MPL also enables a larger number of jobs to work in favor of lowering their waiting times within the system. Overall, in this experiment the average job response time for each scheme decreases with increasing MPL. The PROC schemes provide better performance than the other schemes at all MPL levels.

Lower MPL limits the PROC schemes in terms of harnessing the gains that result from better decisions regarding boost conflicts. The PROC schemes reduce the average job response time by the relatively small factor of up to 18.8% compared to the other schemes when MPL is equal to two. As shown in Fig. 10b, the system is underutilized for lower MPL, with more time spent in a real-idle component. Moreover, as each node has at most two available processes, the opportunities to apply our reordering function become limited, as indicated by the BCR values in the first column in Table 4. Such small amounts of competition in each node also raises difficulties in finding alternate processes when processes are blocked. Thus, the performance gap between blocking-based and spinning-based schemes is less pronounced at a lower MPL.

With a larger MPL, a larger number of jobs are simultaneously accommodated in the system, which increases the likelihood of experiencing a boost conflict at each context switch (see the BCR values when MPL goes up from 2 to 5 in Table 4). Thus, a scheduling sequence of conflicting processes is frequently affected by the reordering function in each node under large MPL. In addition, for a given communication intensity, a larger MPL increases the number of conflicting processes on average at each boost conflict. This causes the chances for finding a successful candidate among all conflicting processes to be limited in existing ICS schemes. Accordingly, when MPL is increased, the PROC schemes with reordering technique provide the best scalable improvement in the resulting response times among all schemes. This improvement and superior can be also inferred from the contrast in the time spent in spinning and idle components at lower and larger MPLs (see Fig. 10b).

5.4 Effect of task skewness

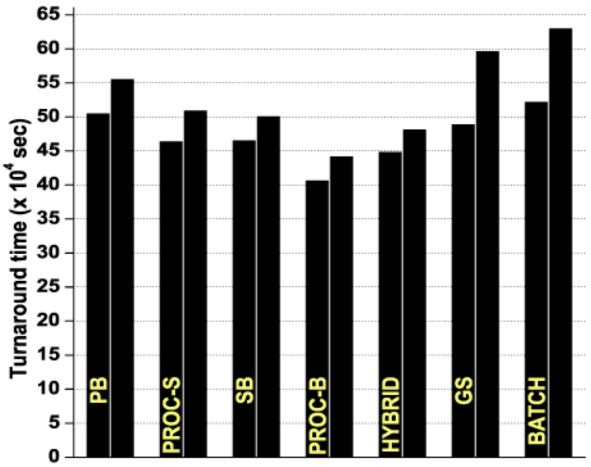
Task skewness within a job can affect the amount of time that a receiving process spins or blocks for a message. To clearly evaluate the effect of task skewness on job execution time, we use turnaround time as a performance metric for this experiment. Here, jobs are started simultaneously, and MPL is fixed to 5. Figure 11a, b, and c show the turnaround times, performance profiles, and the BCR and FDR values for different scheduling schemes with two different skewness values (0% and 60%) for *wl*4.

A greater task skewness prolongs the turnaround time for all schemes, mainly due to the increment of the spinning and idle time by a mismatch of send/receive operations (see Fig. 11a and Fig. 11b). We also observe that GS and BATCH are unable to tolerate this detrimental effect compared to the other ICS schemes because of a lack of overlapped executions among local processes. PROC-S and PROC-B remain robust even at high task skewness, reflecting a more efficient utilization of CPU resources than the rest ICS schemes.

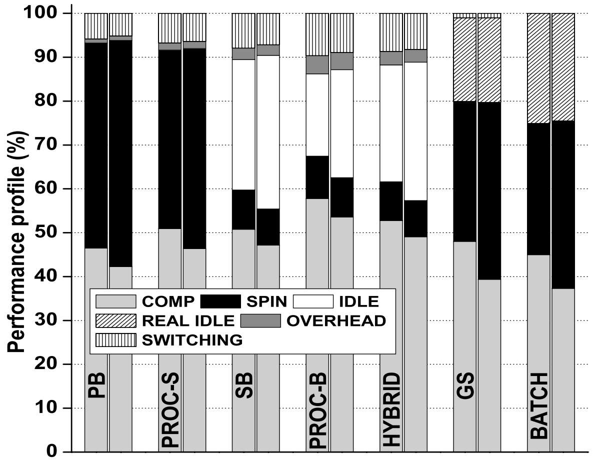
Note that even when jobs are completely balanced, uneven contention can exist among nodes because each job has different node requirements, different communication intensities and patterns, as evident from the BCR and FDR values when $v = 0\%$ in Fig. 11c. With a higher task skewness, incoming messages are more dispersed on average over a period of time in each node; consequently this reduces the likelihood of boost conflicts. This is true for blocking-based schemes, however, in spinning-based schemes, unnecessary spinning for a message (that has not yet arrived) causes the BCR values to increase even further with increasing task skewness (see the BCR values for PB and PROC-S when $v = 60\%$ in Fig. 11c). For this reason, with a higher task skewness, PROC-S shows marginally better performance improvement over PB than that at a lower degree of skewness, while PROC-B shows slightly less improvement but provides the shortest turnaround time.

5.5 Fairness

It is valid to question whether the PROC schemes hurt fairness to different types of jobs in achieving performance gains. To investigate this issue, we ran workload *wl*5 where the jobs are classified as one of three different types: CPU (J1), communication



(a) turnaround time



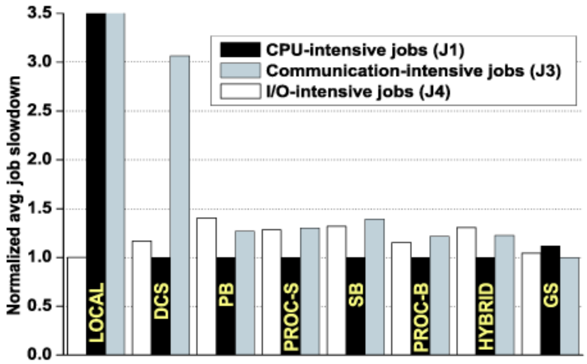
(b) performance profile

	$v = 0\%$		$v = 60\%$	
	BCR	FDR	BCR	FDR
PB	65.98	72.76	66.30	73.43
PROC-S	66.89	35.12	68.63	36.09
SB	45.98	62.43	40.17	61.12
PROC-B	44.93	31.32	38.84	28.66
HYBRID	44.16	56.38	37.86	55.94

(c) BCR and FDR

Fig. 11 Impact of task skewness (*wl*4, $v = 0\%$ (left-hand bars) and 60% (right-hand bars); MPL = 5)

(J3), and I/O-intensive (J4). In a similar way to that reported by [31], we calculated the coefficient of variation in slowdown over three different types of jobs to examine the degree of bias of each scheduling scheme based on the nature of jobs. A smaller



(a) normalized avg. job slowdown

Scheme	Fairness
LOCAL	1.0743
DCS	0.5371
PB	0.1369
PROC-S	0.1153
SB	0.1374
PROC-B	0.0803
HYBRID	0.1105
GS	0.0456

(b) coefficient of variation of slowdown

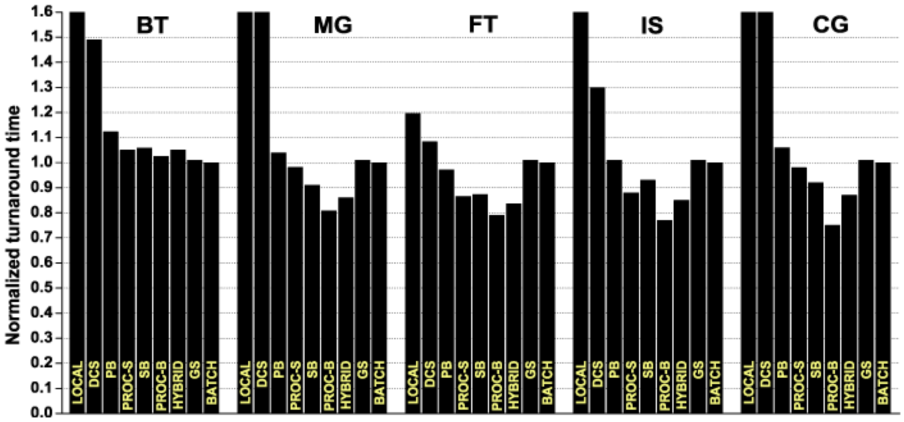
Fig. 12 Results from fairness experiments (*wl*5, MPL = 5, $v = 0\%$)

variation indicates a fairer scheme. Figure 12 shows the average job slowdown (normalized with respect to the job type with the least slowdown) and the degree of fairness for each scheme. First of all, we can see that GS is the most fair with the lowest coefficient of variation (all three types of jobs experiencing almost equal slowdowns) as expected. LOCAL runs counter to fairness in that communication-intensive jobs have the largest slowdown compared to the other schemes. Of the other schemes, PROC-B, HYBRID, and PROC-S provide better fairness, with a low variation of slowdowns. PB and SB fall in the next category followed by DCS.

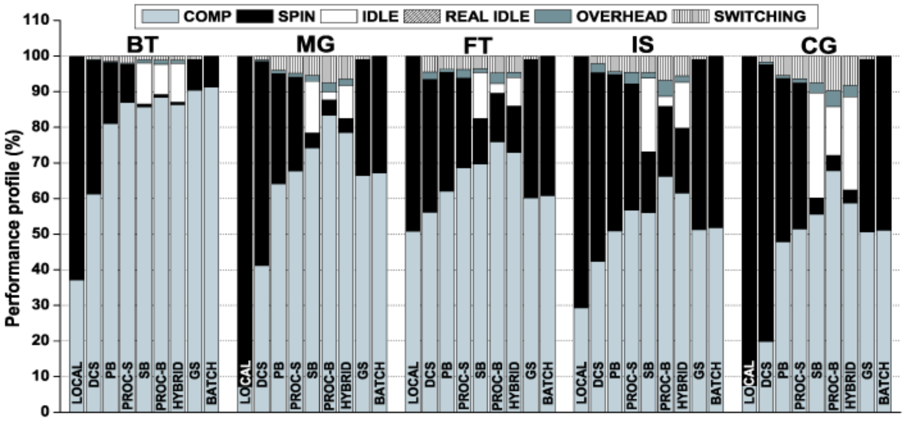
5.6 Realistic workload performance

The workloads evaluated thus far have been relatively simple and synthetic. For more realistic workloads, therefore, we consider five parallel applications: BT, MG, FT, IS, and CG. As outlined in Table 2, the chosen applications have different characteristics in terms of computation granularity, communication pattern and topology, and message size distribution. First, we examine the performance of all scheduling schemes for each realistic application. In this experiment, each workload consists of hundreds of identical jobs, where all jobs require 64 nodes and are started simultaneously. We fixed MPL to 5, and v to 10%.

Figure 13a and Fig. 13b show the normalized turnaround times with respect to BATCH and performance profiles of all considered schemes for the realistic workloads, respectively. From the figures, we clearly confirm that with the use of reordering technique, PROC-B and PROC-S always outperform other blocking-based and spinning-based alternatives, respectively, for all cases of realistic workloads. Of these, PROC-B performs the best across all workloads, and consistently shows superior speedup to BATCH. LOCAL and DCS show much lower resource utilization than the other schemes. Note that since all jobs in each realistic workload require entire nodes in the system, GS gives comparable turnaround time with BATCH, without incurring external resource fragmentation as shown in Fig. 13b. For applications with low communication intensity (such as BT), the turnaround time difference between PROC and other schemes is minimal. However, as the communication intensity increases



(a) normalized turnaround time with respect to BATCH



(b) performance profile

Fig. 13 Performance comparisons of all scheduling schemes for each realistic workload (MPL = 5, $v = 10\%$)

(BT < MG < FT < IS < CG), the performance gain of the PROC schemes over other schemes becomes more distinguished. PROC-B and PROC-S show speedup as high as 25.6% and 13.5%, respectively, compared to BATCH. The results of this experiment also reconfirm that blocking-based schemes (PROC-B, SB, and HYBRID) outperform spinning-based schemes (DCS, PB, and PROC-S).

We conducted another set of experiments using job mixes from the above five applications. Here, we imitated the arrival pattern of jobs in a real system by randomly generating several jobs with different computation granularity, communication intensity, and pattern. We assumed that these jobs arrive at the cluster with exponentially distributed inter-arrival times. For this experiment, we created a series of workloads with different job-arrival rates by multiplying job arrival times by a constant.

Figure 14 shows the performance changes in different scheduling schemes for the mixed workloads as the job-arrival rate increases. In this experiment, we exclude LOCAL and DCS because there is no point to show their performance. As expected,

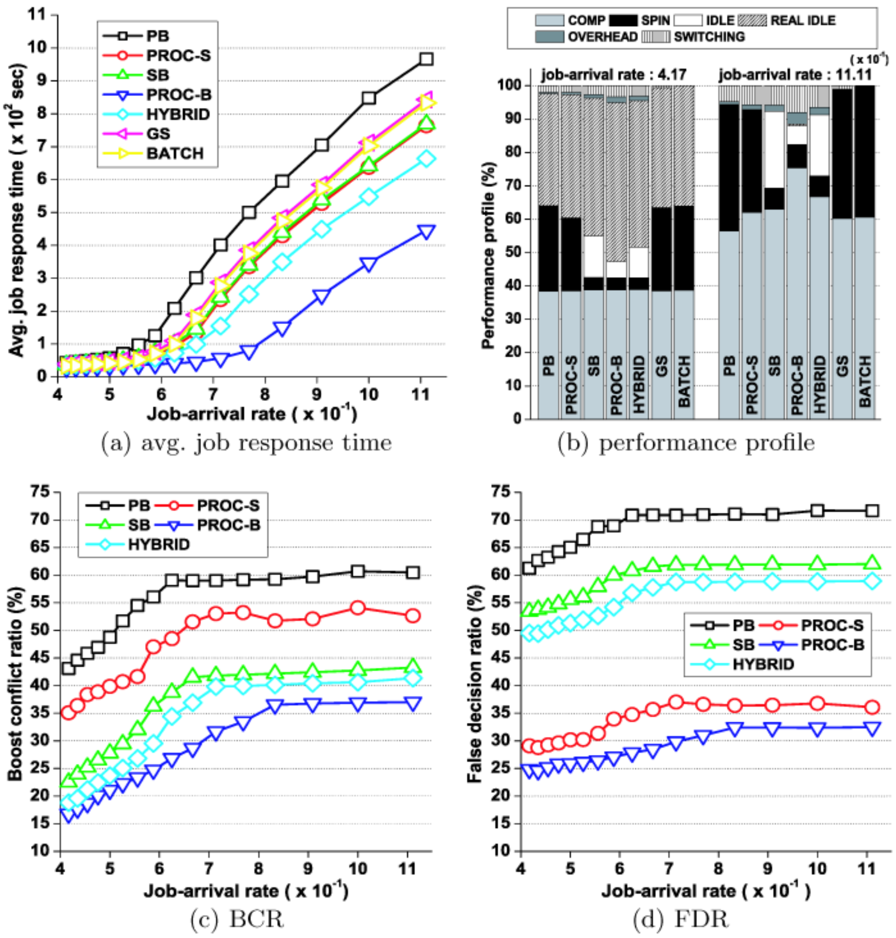


Fig. 14 Performance of all scheduling schemes for the mixed realistic workloads (MPL = 5, $\upsilon = 10\%$)

from Fig. 14a, we observe near-linear growth in response times with increasing arrival rate for all schemes. Again, PROC-B shows the most competitive performance, yielding much lower response times than other schemes, while PROC-S shows comparable performance with SB. With a lower job-arrival rate, a newly arrived job at the cluster system can start almost immediately. This makes MPL as low as one or two, which in turn allows less room to express the differences among seven schemes (PB, PROC-S, SB, PROC-B, HYBRID, GS, and BATCH). However, as shown in Fig. 14c, the increment of the job-arrival rate introduces more competition on the shared CPU at each node and generally increases the probability of a boost conflict. In this situation, wasted spinning and idle time is significantly reduced by using the PROC schemes that recommend the best candidate to the native scheduler (see Fig. 14b and 14d). For an arrival-rate equal to 11.11, for example, it is noticed that PROC-B shows only about 10% false decisions over the total number of context switches. This sophisticated handling of conflicting processes enables more jobs to be pushed at earlier times and leads to shorter average job response times.

6 Concluding remarks

Implicit coscheduling (ICS) has been demonstrated to be an effective technique in enhancing the performance of parallel applications in multi-program-med clusters. However, one major problem found in the existing ICS schemes is that they do not incorporate any steps to attempt to properly handle the priority boost conflict. This results in a harmful scheduling sequence of conflicting processes, thereby interfering with effective utilization of underlying system resources.

In this paper, we proposed the exploitation of the runtime difference in contention across remote nodes to address this problem. We also demonstrated an innovative coscheduling scheme that adaptively regulates the scheduling sequence of conflicting processes on the basis of the measured minimal rescheduling latency of their correspondents. To the best of our knowledge, no previous study has exhaustively investigated this issue in the context of ICS combined with contention on multi-programmed clusters.

To verify the importance of resolving boost conflicts and analyze the performance impacts of the proposed reordering technique, we performed a broad range of experiments using several workloads. The results provide a number of interesting insights:

1. Priority boost conflict is common in multi-programmed clusters that deploy ICS mechanisms, and should therefore be carefully handled to improve system utilization. For example, spinning-based (or blocking-based) schemes experience as high as about 78.9% (or 52.8%) boost conflicts over the total number of context switches under communication-intensive workload.
2. Significant performance improvement is achieved with the proposed PROC schemes. The main reason for this improvement is that PROC seeks to avoid unnecessarily wasted spinning and idle time by attaining a marked reduction in false decisions upon the boost conflict. This performance gain is even more pronounced when a communication-intensive workload and/or a larger MPL are applied to the system.
3. We recommend the use of blocking-based schemes (PROC-B, HYBRID, and SB) rather than spinning-based schemes (DCS, PB, and PROC-S) since the former techniques consistently outperform the later ones. There is little reason to use BATCH, as other ICS schemes such as PROC-B, HYBRID, SB, and PROC-S provide better performance. In addition, to some extent, BATCH and GS can suffer from both internal and external resource fragmentation, and they do not have much scope for tolerating task imbalance as other ICS schemes.
4. The proposed PROC-B is the best performer among all considered schemes over a range of different workloads in terms of response time and overall system throughput, without compromising fairness. Our results show that PROC-B reduces the average job response time by up to 50.4% compared to SB and by up to 72.5% (or 66.8%) compared to BATCH (or GS) under communication-intensive workload.

Given the excellent performance behavior of PROC schemes highlighted in the present study, future work will be directed towards investigating other PROC heuristics with minimal overhead and expanding upon our findings by employing memory-aware and backfilling [16] allocators. We are currently working on developing a prototype PROC in MPI [17] environments over Linux. Then, we also plan to conduct performance comparisons of PROC with gang scheduling implementations such as FCS [10] and Score-D [13] on large-scale and heterogeneous Linux cluster platforms.

References

1. Agarwal S, Choi GS, Das CR, Yoo A, Nagar S (2003) Coordinated coscheduling in clusters through a generic framework. Clust Comput (Dec):84–91
2. Anderson TE, Culler DE, Patterson DA (1995) A case for now (networks of workstations). IEEE Micro 15(1):54–64
3. Anglano C (2000) A comparative evaluation of implicit coscheduling strategies for networks of workstations. In: Proc of the 9th IEEE international symposium on high performance distributed computing, 2000, pp 221–228
4. Borden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su W (1995) Myrinet: a gigabit-per-second local area network. IEEE Micro 15(1):29–36
5. Choi GS, Kim JH, Ersoz D, Yoo A, Das CR (2004) Coscheduling in clusters: is it a viable alternative? In: Proc of the 2004 ACM/IEEE conference on supercomputing, Nov 2004.
6. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C (1998) The virtual interface architecture. IEEE Micro 18(2):66–76
7. Dusseau A, Arpaci R, Culler D (1996) Effective distributed scheduling of parallel workloads. In: Proc of ACM SIGMETRICS conference, May 1996, pp 25–36
8. Etsion Y, Feitelson DG (2001) User-level communication in a system with gang scheduling. In: Proc of the international parallel and distributed processing symposium, 2001, p 58
9. Feitelson D (2003) Metric and workload effects on computer systems evaluation. IEEE Comput 36(9):18–25
10. Frachtenberg E, Feitelson DG, Petrini F, Fernandez J (2005) Adaptive parallel job scheduling with flexible coscheduling. IEEE Trans Parallel Distrib Syst 16(11):1066–1077
11. Frachtenberg E, Feitelson DG, Petrini F, Fernandez J (2003) Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources. In: Proc of international parallel and distributed processing symposium, 2003, pp 625–651
12. Franke H, Jann J, Moreira JE, Pattnaik P, Jette MA (1999) Evaluation of parallel job scheduling for ASCI blue-pacific. In: Proc of the 1999 ACM/IEEE conference on supercomputing, Nov 1999, pp 679–691
13. Hori A, Tezuka H, Ishikawa Y (1998) Highly efficient gang scheduling implementation. In: Proc of the ACM conference on supercomputing, Nov 1998, pp 1–14
14. Jette MA (1997) Performance characteristics of gang scheduling in multiprogrammed environments. In: Proc of the ACM/IEEE conference on supercomputing, Nov 1997, pp 1–12
15. Kim JS, Kim KH, Jung SI, Ha SH (2003) Design and implementation of a user-level sockets layer over virtual interface architecture. Concurr Comput: Pract Exp 15(7-8):727–749
16. Lawson B, Smirni E, Puiu D (2002) Self-adapting backfilling scheduling for parallel systems. In: Proc of international conference on parallel processing, August 2002, pp 583–592
17. Myrinet Inc (2003) MPICH-GM software, Oct 2003. Available from http://www.myrinet.com
18. NAS division. The NAS parallel benchmarks. Available from http://www.nas.nasa.gov/Software/NPB
19. Nagar S, Banerjee A, Sivasubramaniam A, Das CR (1999) Alternatives to coscheduling a network of workstations. J Parallel Distrib Comput 59(2):302–327
20. Nagar S, Banerjee A, Sivasubramaniam A, Das CR (1999) A closer look at coscheduling approaches for a network of workstations. In: Proc of ACM symposium parallel algorithms and architectures, June 1999, pp 96–105
21. Ousterhout J (1982) Scheduling techniques for concurrent systems. In: Proc of the 3rd international conference on distributed computing systems, 1982, pp 22–30

22. Parallel workloads archive. Available from http://www.cs.huji.ac.il/labs/parallel/workload
23. Petrini F, Feng W (2001) Improved resource utilization with buffered coscheduling. J Parallel Algorithm Appl 16(2-3):123–144
24. Rencuzogullari U, Dwarkadas S (2001) Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In: Proc of ACM SIGPLAN symposium on principles and practice of parallel programming, 2001, pp 72–81
25. Sabin G, Kettimuthu R, Rajan A, Sadayappan P (2003) Scheduling of parallel jobs in a heterogeneous multi-site environment. In: Proc of job scheduling strategies for parallel processing, 2003, pp 87–104
26. Schwetman HD (2001) CSIM19: a powerful tool for building system models. In: Proc of the 2001 winter simulation conference, 2001, pp 250–255
27. Sobalvarro P, Pakin S, Weihl B, Chien AA (1998) Dynamic coscheduling on workstation clusters. In: Proc of the international parallel processing symposium, March 1998, pp 231–256
28. SUN Microsystems Inc (1997) Solaris 2.6 Software Developer Collection. Available from http://www.sum.com
29. Zhang Y, Franke H, Moreira JE, Sivasubramaniam A (2003) An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. IEEE Trans Parallel Distrib Syst 14(3):236–247
30. Zhang Y, Sivasubramaniam A (2004) ClusterSchedSim: a unifying simulation framework for cluster scheduling strategies. In: SIMULATION: transactions of the society for modeling and simulation, 2004, pp 191–206
31. Zhang Y, Sivasubramaniam A, Moreira J, Franke H (2001) Impact of workload and system parameters on next generation cluster scheduling. IEEE Trans Parallel Distrib Syst 12(9):967–985

Jung-Lok Yu received the B.S. degree from the Soong-Sil University, Korea, in 1999, and the M.S. and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), in 2001 and 2007, respectively. He is currently with Samsung Electronics Co. Ltd., Korea. His research interests include computer architecture, cluster computing, operating systems, embedded systems, and grid computing.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He was with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. He is currently an associate professor of the department of electrical engineering and computer science at Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, he was a senior member of research staff at Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002. His research interests include flash memory-based storage and operating systems.

Seung-Ryoul Maeng received the B.S. degree in electronics engineering from the Seoul National University, Seoul, Korea, in 1977, the M.S. and Ph.D. degrees in computer science from KAIST in 1979 and 1984, respectively. Since 1984 he has been a faculty member at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include parallel computer architecture, vision architecture, embedded systems, and cluster computing.