IPSJ Transactions on Advanced Computing Systems Vol. 4 No. 2 94–113 (Mar. 2011)

Regular Paper

User-level Enforcement of Appropriate Background Process Execution

Yoshihisa Abe,[†1,†2] Hiroshi Yamada[†1]
and Kenji Kono[†1]

Idle resources can be exploited not only to run important local tasks such as data backup and virus checking, but also to make contributions to society by participating in distributed computing projects. When executing background processes to utilize such valuable idle resources, we need to control them explicitly to avoid foreground performance degradation. Otherwise, the user will be discouraged from exploiting idle resources. In this paper, we show that we can detect resource contention between foreground and background processes and properly control background process execution at the user level, without modifications to the underlying operating system or user applications. We infer resource contention from changes in the approximated resource shares of background processes. In deriving those resource shares, our approach takes advantage of dynamically enabled probes. Also, it takes account of different resource types and can handle multiple background processes with varied resource needs. Our experiments show that our system keeps the increase in foreground execution time due to background processes below 16.9% – often much lower in most of our experiments.

1. Introduction

There has been increasing attention to idle resource utilization, which exploits underutilized resources in the system in order to perform valuable tasks. One popular way of utilizing idle resources is to join open computing projects such as SETI@home [3] and Folding@home [17]. They primarily use computer resources contributed by users on a voluntary basis in order to perform scientific computations. Such distributed computing projects turn idle resource utilization into a new way of social contribution by increasing active resource use. In addition, there exists more traditional, local ways of idle resource utilization for improving performance, robustness, security, and other aspects of the system. Examples include database reorganization and disk layout reconfiguration for improving performance, data backup and replication for robustness, and virus checking and software updating for security.

Although idle resource use provides an opportunity to perform numerous valuable tasks, it does not come for free. General-purpose schedulers used by common operating systems do not sufficiently prevent the interference of background processes with foreground processes. Background processes for idle resource utilization should consume only otherwise wasted resources and should be assigned as little computing capacity as possible that could be allocated to foreground processes. Otherwise, users would not choose to make use of idle resources given the adverse impact on their foreground processes. General-purpose schedulers lack the concept of background processes for idle resource utilization, and consequently they can considerably degrade the performance of foreground processes. Therefore, an explicit mechanism is needed that provides appropriate control over background process execution.

Furthermore, the growing value of idle resource use introduces a new challenge for such a mechanism. As mentioned earlier, projects such as SETI@home and Folding@home rely mainly on computer resources contributed by volunteers. This fact demands that a mechanism for background process control be easily deployable; if it required significant modifications to existing systems, it would not attract users and fail to encourage them to participate in those distributed computing projects. Also, users need an easily deployable mechanism to execute background processes for protecting their computers from various troubles. For example, they should be able to run programs such as those for data backup and virus checking readily using idle resources.

Our goal is to develop a mechanism for exploiting idle resources in the system that effectively prevents the throughput degradation of foreground processes and yet does not require any significant modification to the user's existing environment. In this paper, we argue that we can reasonably infer the interference of background processes with foreground processes at the user level, and properly control the execution of those background processes without modifying either

†1 Keio University
†2 Carnegie Mellon University
This paper is an improved version of the authors' previous work [1].

© 2011 Information Processing Society of Japan

the operating system kernel or user applications. Our proposed approach takes advantage of dynamically enabled probes, and takes into account different types of resources, such as CPUs, disks, and network interfaces, in combination to judge whether to suspend background processes. Also, it can handle multiple background processes with varied resource needs.

One limitation of our approach is that it focuses on minimizing foreground throughput degradation when exploiting idle resources for background process execution; it does not try to prevent the increase in foreground response time. Our mechanism aims to allow improving the overall system throughput by executing beneficial background activities in an unobtrusive manner. Since aggressive resource utilization conflicts with good response time preservation, we do not consider the latter in this work.

The remainder of this paper is organized as follows. In Section 2, we describe the background and related work, along with our motivations. Next, we explain our approach in Section 3, and practical issues in employing the approach in Section 4. Section 5 briefly summarizes our implementation and describes the execution control of background processes. Section 6 presents our experimental results. Finally, we discuss essential aspects of our approach in Section 7 and conclude in Section 8.

2. Background and Related Work

In this section, we briefly describe why we need to explicitly control the execution of background processes to preserve the throughput of foreground processes. We then explain previous approaches to idle resource utilization, and restate our motivation examining these related works against our objectives.

2.1 Insufficiency of General-purpose Schedulers

Most common operating systems use general-purpose schedulers, such as priority-based ones, to prioritize processes. However, such general-purpose schedulers lack the concept of background processes for idle resource utilization and fail to control their execution properly. The primary reason is that most general-purpose schedulers take only CPU usage into account, and do not consider other resources in combination. They thus do not appropriately handle cases in which foreground and background processes compete for other resources, such as disks and network interfaces. Even worse, processes usually have a greater impact on each other when they contend for these types of resources.

In addition, modern process schedulers can exhibit behavior that is intended to achieve general fairness among processes but is undesirable in terms of controlling background processes for idle resource utilization. For instance, priority-based schedulers often change the priorities of processes dynamically for reasons such as avoiding the starvation of low-priority processes. As a result, the operating system does not always execute foreground processes with initial high priorities, and may instead choose to run background processes with dynamically raised priorities. On some operating systems, such as Solaris 10, processes with the maximum nice value are basically scheduled only when no other processes are runnable. Strict prioritization of CPU-intensive processes is thus possible on these systems. However, there exist cases in which CPU-intensive processes with dynamically lowered priorities experience performance degradation, as we will show in Section 6.

For these reasons, background processes for idle resource utilization cannot be appropriately handled by general-purpose schedulers. There is a need for an explicit mechanism that prioritizes foreground processes under any circumstances and allows background processes to be executed only when there exists no active foreground processes.

2.2 Related Work

Different approaches to exploiting idle resources were proposed previously. Idletime scheduling [11] is a kernel-level approach to explicitly prioritizing foreground requests over background ones. It introduces preemption intervals, during which no background requests are served even if no foreground requests exist and, as a result, resources remain idle. A preemption interval amortizes the cost of background request preemption over a series of foreground requests arriving one after another within the interval, preventing foreground throughput degradation. Idletime scheduling can be applied to disk and network scheduling with small amounts of modification to the operating system.

Freeblock scheduling [20),21] processes background requests to a disk in a way that has virtually no performance impact on foreground requests. It determines the positioning time between two successive foreground requests, and schedules

© 2011 Information Processing Society of Japan

and outstanding background request if and only if it can be served during that positioning time. Using detailed information about the underlying disk, freeblock scheduling can significantly improve disk bandwidth utilization.

TCP Nice [24] and TCP-LP [16] provide protocol-level mechanisms for background network data transmission. TCP Nice infers potential network congestion from round-trip packet delays. It reacts to congestion more sensitively and rapidly than TCP Vegas [6] by multiplicatively decreasing the congestion window size when more than a certain fraction of packets sent within a round-trip time frame signifies congestion. Similarly, TCP-LP uses one-way packet delays of a connection to detect early signs of congestion. When it first observes a packet delay indicating congestion, it halves the congestion window and starts a timer. If it catches another sign of congestion before the timer expires it sets the window to one, thereby minimizing interference with other connections.

MS Manners [8] is a user-level approach to controlling the execution of low-importance processes. It decides whether to allow the execution of a low-importance process based on its progress rate. When the progress rate decreases, it assumes that the progress of high-importance processes also slows down. In such a situation, therefore, MS Manners suspends the process of low importance to prevent performance degradation of the more important processes. It uses a statistical method to properly judge if the progress rate of the low-importance process has slowed down.

Open computing projects, such as SETI@home [3], Folding@home [17], and others using the BOINC infrastructure [2] employ a screen saver approach. They start computations after a certain period of time has passed since the last user input. This approach is simple and does not require significant modifications to the underlying operating system or user applications. Also, the user can specify the fraction of resource capacity these projects can receive through preference settings.

Golding, et al. [14] propose a general framework for utilizing idle resources. Their framework consists of three components: *predictor*, *skeptic*, and *actuator*. A predictor outputs pairs of the start time and duration of predicted idle periods. A skeptic takes predictions made by one or more predictors, and improves the accuracy of idle period estimations by eliminating false estimations based on certain criteria. Finally, the actuator controls the execution of background tasks according to these idle period estimations. Golding et al. examine a wide variety of idle time start and duration predictors and skeptics. Although they evaluate the framework by simulation, it would be possible to implement it at the user level with the use of dynamically enabled probes, of which our approach takes advantage.

Idle resource utilization has also been explored at the level of clusters of computers. Condor [19] improves the overall utilization of workstations by placing tasks on idle workstations in a network. When the user returns to a workstation that is executing remote jobs, Condor transfers these jobs to other idle workstations in order to dedicate the workstation to the user. The Stealth Distributed Scheduler [15] preserves the performance of a workstation executing remote jobs that its owner receives by explicitly prioritizing system resources. It implements prioritized virtual memory and file system cache in order to avoid the interference of remotely executed jobs with the workstation owner's local jobs, while exploiting whatever resources not used by these local jobs.

2.3 Motivation

The approaches described above are limited in some ways and are not user-friendly enough to encourage active idle resource utilization. Such limitations can be grouped into four categories.

The first category, significant modification of the user's existing environment, poses a primary challenge we address in this paper. Generally, a mechanism at a low level has access to precise information about the system, and thus can manage resources in a fine-grained manner. For example, Freeblock scheduling achieves its best performance when it is implemented inside disk firmware, rather than at the user level. Idletime scheduling, TCP Nice, TCP-LP, and Stealth are also low level approaches implemented inside the operating system. Although these approaches achieve high efficiency, the fact that they need modifications to the underlying system may discourage users from active idle resource utilization.

Next, some of the approaches do not consider actual usage of resources in a way that is sufficiently fine-grained. The screen saver approach used by open computing projects relies on the assumption that resources are idle when the user is away from the machine, which is often untrue. Also, although the user can

© 2011 Information Processing Society of Japan

specify the amounts of resource capacity BOINC projects receive, this approach cannot flexibly deal with workload changes. Furthermore, usually it is not obvious to the user how much capacity these projects should be assigned in order to maximize idle resource use while avoiding the degradation of foreground activity performance. Condor performs preemptive transfers of remote jobs and prevents them from staying at a workstation used by its owner; thus, it would not allow them to consume unused resources in a fine-grained fashion.

Third, works such as Freeblock scheduling, TCP Nice, and TCP-LP target particular resource types. They control the usage of specific resources with specialized mechanisms that are not applicable to other kinds of resources. For a background process for idle resource utilization to execute without frustrating the user, however, we need to consider different resources in combination. For example, a background process may perform intensive computation in one phase and write the results to a disk in another phase. Analyzing either CPU or disk contention alone is not enough to sufficiently detect the adverse impact of such a process on foreground processes.

Finally, an effective approach to idle resource utilization needs to handle varied foreground and background workloads. MS Manners observes the progress rates of processes rather than directly considering resource usage. As a result, it has limitations such as requiring to know the base progress rates of low-importance processes in advance and allowing the execution of only one low-importance process at a time. These limitations prevent MS Manners from dealing flexibly with diverse workloads. Also, although the framework proposed by Golding et al. could incorporate other idle period predictors, they suggest resource idleness be detected using heuristics such as the arrival rate of disk I/O requests and the time that has passed since the last I/O request. The efficacy of these heuristics depends on the workload and configuration being considered. Consequently, appropriate metrics for predicting idle periods need to be determined by means of some off-line procedure, or have to be replaced or modified on-line based on some learning process, which is likely to limit the ability to adapt to workload changes flexibly.

In order to encourage users to exploit their underutilized resources, we need to address those four issues described above. Our motivation is to provide a mechanism for efficiently controlling background processes that (1) is easily deployable, (2) reflects actual resource usage, (3) is applicable to more than one resource type, and (4) manages varied background workloads effectively.

3. Approach

To address the issues described in the preceding section, we propose a user-level approach to controlling background process execution. Specifically, it aims at providing a system-wide solution that manages background processes specified by the user. Driven and guided by our system design objectives, we estimate at the user level the usage of resources using indicative system information, and determine whether to suspend background activities based on the derived resource usage and a conservative assumption.

Two of our objectives – demanding no significant modification to the user's environment and reflecting actual resource usage in a fairly fine-grained manner – led us to speculate on the resource usage of processes at the user level. At the user level, it is difficult to obtain the precise knowledge of resource usage. We instead estimate resource usage by using certain statistical information that is readily available from outside the operating system (e.g., the number of disk blocks read by a process). In obtaining such system information, we take advantage of dynamically enabled probes [7),22),23)]. Previous works [4),5),12)] suggest the benefits of exposing a certain level of operating system information to the user level. Dynamic instrumentation of operating systems with probes is an active area of research, and one of the most widely accepted ways of enabling such exposition. It allows numerous kinds of system information to be detected on the fly from running operating systems, while having a negligible impact on system performance when turned off. Exploiting those probes, which are becoming commonly available on modern operating systems [9),10),18)], leads to a more general approach to our objectives than those proposed in the past that involve modifications at the operating system level.

The other two goals, accounting for different types of resources and dealing with varied workloads, resulted in our method of inferring resource contention by using approximated resource shares of background processes. To judge whether to suspend background processes under different circumstances, we need a gen-

© 2011 Information Processing Society of Japan

eral criterion based on resource usage for determining that they interfere with foreground activities. Our approach is to use the resource shares of background processes that are derived from the statistical information mentioned above. If a background resource share is low, we decide that contention for the corresponding resource exists between foreground and background processes. (For brevity, we refer to resource contention between foreground and background processes as "resource contention" or just "contention" throughout this paper.) We then suspend the background activities, expecting that the foreground processes will consume the resource capacity reclaimed from them. In other words, we conservatively assume that those background processes have "stolen" resources from foreground processes and caused foreground throughput degradation. To judge if an approximated background resource share is low enough to suspend background activities, we use a threshold over the share.

The basic approach described above poses practical questions such as (1) what statistics we can use to derive approximated resource usage, (2) what processes we should or should not consider to obtain meaningful resource usage, (3) if we can always rely solely on relative resource usage of foreground and background processes, and (4) how we can decide appropriate thresholds over the background resource shares. We will explore these issues in the next section.

4. Practical Issues

In this section, we review fundamental issues that must be addressed for our approach to work in practical situations. Discussions in this section are based on our experience of employing the approach on Solaris 10. We, however, expect that they are general enough to be applicable to other common platforms.

4.1 Resources and Corresponding Statistics

We need to obtain system information indicative of resource usage at the user level, where available information is limited compared to the inside of the operating system. In addition, we have to keep the information to analyze simple as we need to process it frequently in order to rapidly respond to system workload changes. In this work, we take into account three types of resources: CPUs, disks, and network interfaces. To estimate the usage of these resource types, we use the following statistics.

- CPUs: cumulative time for which processes are scheduled on them.
- Disks: the number of blocks read or written synchronously by processes.
- Network interfaces: the number of times processes call write() or send() with associated descriptors.

Those statistics do not represent the exact resource usage of their corresponding resource types. However, as confirmed by the effectiveness of our system shown in our experimental results, they adequately reflect resource usage and serve as useful information on which our decision on background process execution can be based. We could use more detailed system information that represents resource usage more precisely than the simple statistics above, but such information usually results in larger overheads caused by the required probes. Therefore, we use fairly simple statistics to represent resource usage sufficiently for our purpose while still causing a small overhead.

Also, note that we do not consider asynchronous disk I/O and inbound network traffic. Asynchronous disk requests cannot be associated completely with the processes that have issued them. Considering them would thus lead to complicated and incorrect background disk share estimations. As for inbound network traffic, we do not know if suspending background traffic truly improves foreground performance. If background traffic does not go through the bottleneck of foreground traffic, suspending it would only decrease the total inbound throughput without improving foreground throughput. For these reasons, we exclude asynchronous disk I/O and inbound network traffic from consideration.

For each resource type, we obtain the approximated resource share of background processes by calculating the proportion of the statistical values associated with them to the system total. This calculation is represented as:

$$BGShare = \frac{BG}{SystemTotal} \quad (1)$$

BGShare is the background resource share, *BG* the statistics value of the background processes, and *SystemTotal* the statistics value of all processes in the system. *BGShare* approximates the fraction of the resource capacity allocated to the background processes.

4.2 Ignoring Certain Processes

Some types of processes obscure the direct relationship between the resource

© 2011 Information Processing Society of Japan

shares of the user's foreground and background activities. These are the ones that persistently exist in the system and consume resources but do not specifically represent the user's foreground work. If we include their statistics in the system totals when calculating *BGShare*, they will consistently make its value lower. To consider their effect on *BGShare*, let us rewrite Eq. (1) as follows:

$$BGShare = \frac{BG}{FG + BG + SystemServices} \quad (2)$$

FG and *SystemServices* are the statistics values of foreground processes and of those processes that are neither foreground nor background, respectively. *SystemServices* in the equation makes smaller and less clear the difference in background resource shares due to the existence of resource contention. Consequently, it will be less easy to detect resource contention.

A primary example of those processes categorized as *SystemServices* is the swapper process (which is the scheduler process *sched* on Solaris 10). When a CPU is not busy, a large part of its cycles is assigned to the swapper, in which case it often does not perform any beneficial task for the system. Other examples include the X server and those processes related to dynamically enabled probes. If an X server is present on the system, it runs while consuming a fraction of the system resources, whether or not the user's foreground or background processes are executed. Processes related to probes exist while system information regarding both foreground and background activities is collected. They thus do not directly represent the user's activities. We intentionally ignore the statistics related to these kinds of processes to better estimate the existence of resource contention.

4.3 Handling CPU Contention

CPUs have different factors to consider, compared to those of disks and network interfaces, at both single- and multiple-instance levels. We employ two improvements to our basic approach in order to address such aspects and handle CPU contention effectively.

4.3.1 Considering CPU Idleness

For each resource instance, our approach to inferring resource contention considers the direct relationship between foreground and background resource consumption. It assumes that suspending background processes when their relative resource usage is low will let foreground processes consume more resource capacity than currently allocated. For disks and network interfaces, this expectation is justifiable because background requests to these resources can have significant impacts on concurrent foreground requests. However, CPUs differ from these peripheral devices; given a particular portion of the CPU capacity assigned to a background process, we need to know if the allocated resource is all the process needs or is a result of contention with other processes. A background process with a low CPU share should be suspended when it really competes with foreground processes. Otherwise, it should be allowed to run.

To judge if a background process with a low CPU share should really be suspended, we use the approximated CPU share of the swapper process as an indication of CPU idleness. When *BGShare* of a CPU is low, we additionally check if the swapper share is lower than a threshold. If it is, we conclude that background activity suspension will let foreground processes be assigned more CPU capacity; otherwise, we allow background process execution.

We have found that considering CPU idleness does improve our approach, and also that the performance of our method is relatively insensitive to the exact value of the threshold over the swapper share. Based on this observation, we currently set the threshold to 25% as it has been shown to work well.

4.3.2 Handling Multiprocessors

When considering multiple instances of the same resource type, CPUs differ from disks and network interfaces in that processes can basically use any of them. For disks and network interfaces, processes need to send a request to a specific device that holds data being accessed or that is connected to an appropriate network. Thus, we can analyze *BGShare* per device and suspend only those background processes that compete for the same devices used by foreground processes. Although this basic approach can still be used for multiprocessors, we need to address certain behavior of the underlying scheduler to guarantee strict process prioritization. **Figure 1** illustrates this behavior by showing foreground and background processes in a multiprocessor environment. The left case of the figure shows a simple situation with resource contention. In this case, we need to suspend BG1 to guarantee that FG1 receives as much CPU capacity as possible. This decision can be made in a straightforward manner by observing *BGShare*

© 2011 Information Processing Society of Japan

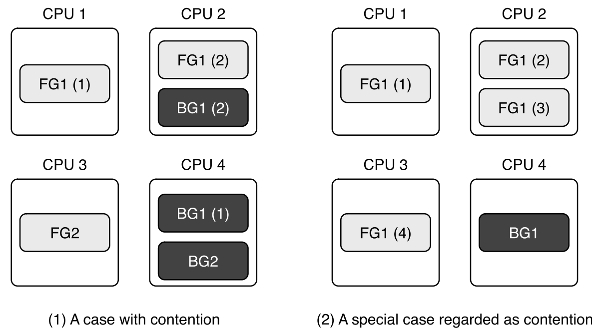


Fig. 1 CPU contention in a multiprocessor environment. "FG" and "BG" indicate foreground and background processes, respectively, followed by a number for process identification. Numbers in parentheses distinguish threads of multi-threaded processes. For simplicity and clarity, only active user processes are depicted in the figure and other insignificant processes are omitted.

of each CPU.

The right case in Fig. 1 shows an example situation in which the basic approach alone is not sufficient. In this case, the underlying scheduler prioritizes foreground process FG1 over background process BG1 by assigning more CPUs to the former than to the latter. When the scheduler behaves in this manner, we cannot solely rely on *BGShare* to infer contention, because BG1's share of CPU4 does not decrease due to the existence of FG1. Still, FG1's performance is affected because its four threads are run only on three CPUs. We therefore need to regard such a case as an instance of resource contention caused by background processes. To address this problem, we conservatively suspend background processes when all CPUs are actively utilized, because they might be interfering with the foreground processes' performance. In other words, only when at least one CPU remains idle can we conclude that foreground processes have not been forced into running on a limited set of CPUs and allow background processes to run.

4.4 Detecting Resource Contention

This section describes how we determine the threshold over *BGShare* of each resource type, which we use to infer the contention of the corresponding resource. The data shown in this section were collected on our test machines, each with a 2.4 GHz Pentium 4 processor, 512 MB of memory, and a 40 GB 7200 RPM disk. For network measurements, a pair of these machines were connected directly through gigabit Ethernet interfaces.

4.4.1 Methodology

We took an empirical approach to finding an appropriate range of thresholds over *BGShare* of each resource type. We used two programs to obtain actual statistical information in cases with and without resource contention. One program is for obtaining disk statistics, and the other is for network statistics. The disk program touches the first byte of contiguous 8 KB regions of a 2 GB file. The network program sends data through a TCP connection to a destination node, which just discards the received data. It uses TCP because protocols that establish logical circuits are usually preferable for network applications used for background execution. Depending on given parameters, these two programs spend part of their CPU time simply consuming it in loops, so that they approximately use a specified percentage of the maximum available bandwidth of their corresponding resources. We refer to this percentage as the "resource intensity" of the programs.

To obtain the disk statistics under varied workloads, we ran our disk access program changing its resource intensity. For the case with resource contention, we ran one foreground process with a default priority and one or more background processes with the maximum nice value. These background processes made the foreground process run longer as they sent requests to the same disk, which confirms the existence of resource contention. We fixed the number of foreground processes to 1 because when more of them exist, *BGShare* is usually lower and thus it is easier to judge resource contention. For the case without resource contention, we ran only background processes. The resource intensity of both foreground and background processes was varied from 12.5% to 100% in increments of 12.5%. To obtain the network statistics, we did the same measurements using our TCP program.

We applied different thresholds over the obtained statistics in order to examine how accurately these thresholds infer the existence of resource contention. We define the accuracy of a set of thresholds (each over a different resource type) as the product of two figures. One is the fraction of the statistics samples for which we can correctly judge that contention exists based on the set of thresholds. The other is the fraction of the samples for which we can correctly decide that

© 2011 Information Processing Society of Japan

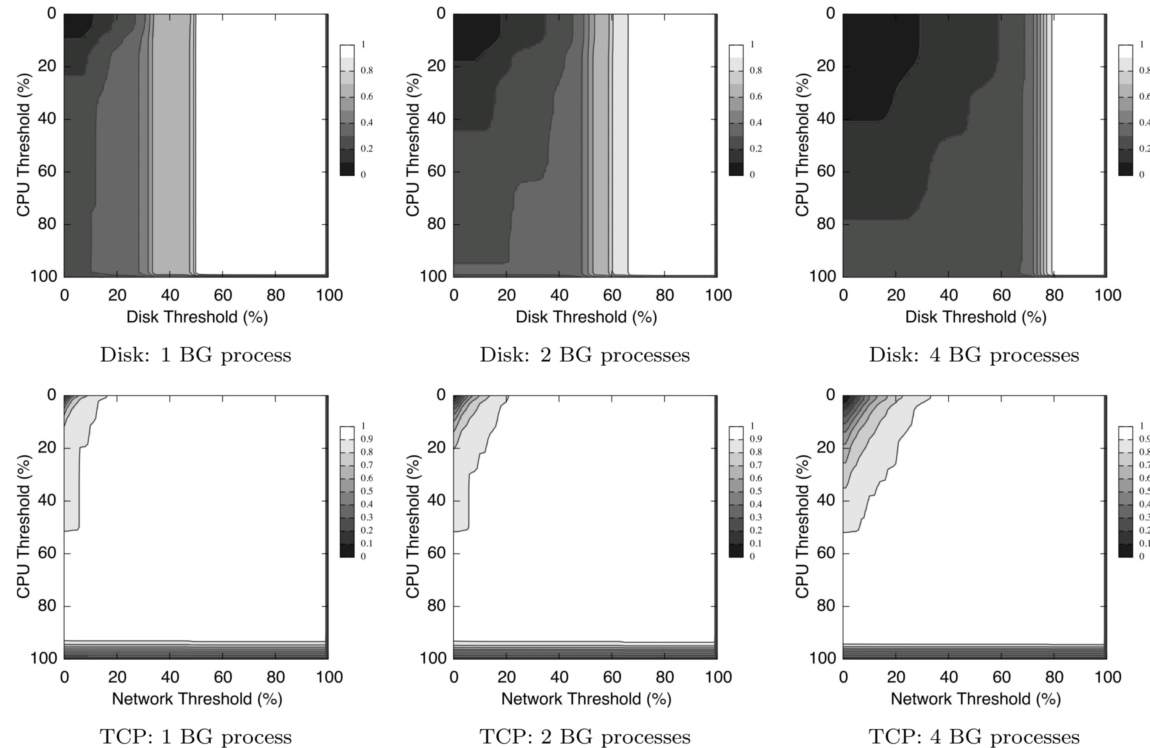


Fig. 2 Accuracy of different thresholds. The top row shows the accuracy of detecting competing processes of our disk access program for different CPU and disk thresholds. Similarly, the bottom row shows the accuracy of detecting competing processes of our TCP program for different CPU and network thresholds. The interval used for aggregating statistics is 1.25 seconds.

contention does not exist. For example, if a certain set of thresholds correctly detects resource contention 80% of the time and correctly concludes the non-exisitence of contention 90% of the time from those samples, its accuracy is 0.72. The intuition behind this definition is the geometric mean of the two numbers squared to make differences easier to observe.

4.4.2 Determining Thresholds

Figure 2 shows the accuracy of different thresholds for varied number of background processes, with dark regions representing low accuracy and light regions high accuracy. For clarity, we show the results of detecting disk and network contention separately. The effect of varied thresholds over the background CPU share is shown for both of the disk and network cases. The figure indicates that wide ranges of thresholds result in very high accuracy. For disk contention, the threshold over the background disk share primarily affects the accuracy, and the threshold over the CPU share has minor effects when the disk threshold is low. For network contention, the CPU threshold has some impact on the accuracy. Specifically, the graphs show that CPU thresholds close to 100% react to fluctua-

© 2011 Information Processing Society of Japan

tions in *BGShare* too sensitively, resulting in lower accuracies. Overall, the large regions of high accuracy indicate that the effectiveness of our approach is fairly insensitive to the exact values of the thresholds, as long as they exist within these regions.

Within the ranges of thresholds with the best accuracy, we selected higher values in order to prevent foreground performance degradation strictly. The thresholds chosen roughly fall into a range between 80% and 90% for CPUs, and are just below 100% for disks and network interfaces. We selected thresholds using combinations of up to 6 different background processes to perform our sample statistics analysis. For larger number of background processes, we simply chose the same thresholds we used for 6 background processes since the appropriate threshold values based on our analysis were fairly stable.

Note that we regard *BGShare* being zero as an indication of no resource contention. Strictly speaking, when *BGShare* is zero, it may be because background processes do not need the corresponding resource, or because contention has prevented them from consuming the resource. However, in the latter case, background processes have not taken away any resource capacity from foreground processes. Thus, keeping these background processes executed is a sound approach. Always assuming the existence of contention given zero *BGShare*, on the other hand, would not work because it unnecessarily prevents the execution of background processes that do not need the corresponding resource. Also, this issue of time periods with no background resource allocation under contention is further addressed by appropriate statistics aggregation, as described below.

The data shown in Fig. 2 were obtained using the sample statistics aggregated for 1.25 seconds. This interval for aggregating statistical information, as well as the thresholds, affects the accuracy of detecting resource contention. **Figure 3** shows the accuracy of detecting contention with the best set of thresholds for varied aggregation intervals. Basically, a short interval is preferred because it allows rapid reaction to workload changes. However, if the interval is too short, fluctuations in *BGShare* are caught too sensitively. In particular, a *BGShare* of zero can be observed frequently under resource contention, due to the underlying scheduler prioritizing foreground processes over background ones. The aggregation interval must be long enough to absorb such fluctuations and appropriately

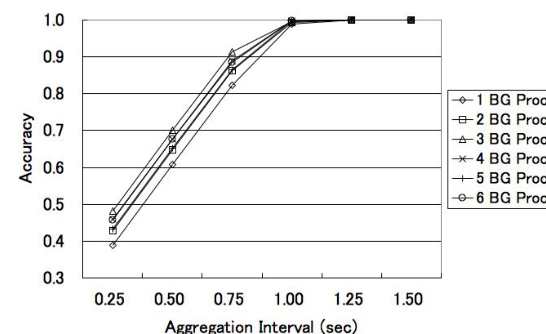


Fig. 3 Interval for aggregating system statistics. The figure shows the accuracy of inferring resource contention for different lengths of statistics aggregation intervals and number of background processes. The x-axis indicates the length of the interval, and the y-axis shows the accuracy of inferring resource contention. Accuracy reported in the figure is the product of the accuracy of detecting disk contention and that of network contention.

judge whether to allow background execution. Figure 3 shows that the interval length of 1.25 seconds is long enough to achieve very high accuracies, regardless of the number of background processes. We selected the interval length based on this observation.

5. System Details

In this section, we first describe our implementation. Next, we explain how our system controls the execution of background processes upon detecting resource contention.

5.1 Implementation

We implemented a daemon on Solaris 10 that observes and controls background processes using the approach described in previous sections, as illustrated in **Fig. 4**. The user executes an application as background work for idle resource utilization by passing the command as arguments to our client program. The client first notifies the daemon of its process ID, sets its own nice value to the maximum defined by the system, and then replaces its own process image with that of the specified application. The daemon controls the execution of background processes by sending signals.

© 2011 Information Processing Society of Japan

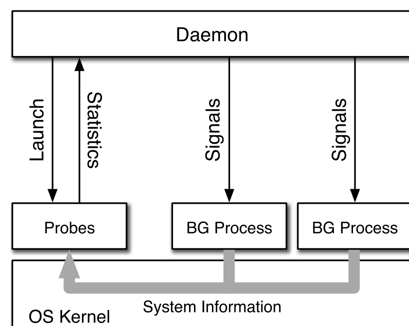


Fig. 4 System implementation. The figure shows the structure of our system. Our client program (not shown in the figure) replaces its process image with that of the specified background application after notifying the daemon of its process ID.

We used DTrace [7] in order to obtain the statistics indicative of resource usage. The number of disk blocks a process reads or writes synchronously is obtained with *io:::wait-done* probe. The time during which a process is scheduled on a CPU is obtained by reading and saving timestamp values [⋆1] inside *sched:::on-cpu* and *sched:::off-cpu* probes. Finally, the number of times a process calls write() or send() is counted using *syscall::write:entry* and *syscall::send:entry* probes. These statistics are sent to the daemon periodically with associated information such as process IDs and file descriptors. The daemon processes these statistics to obtain the background resource shares.

5.2 Background Process Execution

Our system suspends background processes when *BGShare* of any of the three resource types falls below the corresponding threshold, indicating the existence of contention for the resource. When the background processes are suspended, the system needs to determine when those processes can be resumed. We developed two algorithms for this purpose, the Exponentially Increasing Interval (EII) algorithm, which borrows ideas from MS Manners [8], and the Idle Period Detection (IPD) algorithm.

⋆1 We used the *timestamp* variable for the simplicity and ease of implementation. Alternatively we could have used the *vtimestamp* variable, which provides the virtual CPU time while excluding system overheads and thus could be more appropriate.

5.2.1 The Exponentially Increasing Interval Algorithm

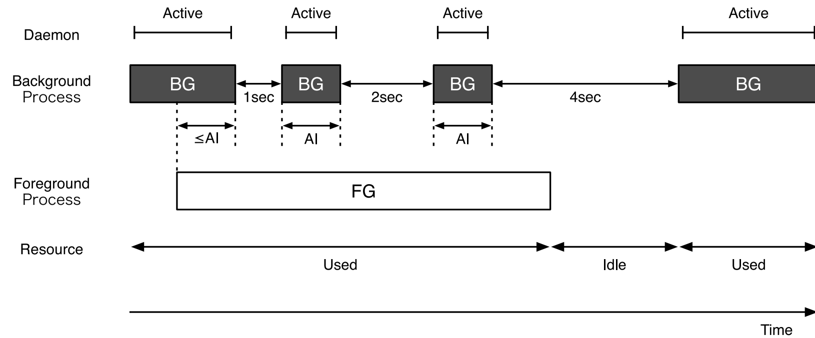
The first algorithm we implemented, called the EII algorithm, repeatedly executes background processes for a short period in order to find out if resuming them causes resource contention with foreground processes. The top part of **Fig. 5** illustrates how the algorithm works. Once background processes have been suspended, the EII algorithm first re-runs them after a small amount of time. It keeps them executed temporarily until enough statistics are collected to judge the existence of resource contention. Then, if the background processes are still contending with foreground activities, the algorithm suspends them for an interval twice as long as the previous one. In this way, the suspension interval grows exponentially until it reaches a pre-defined maximum length, as long as resource contention exists. When resource contention disappears, the background processes are allowed to run continuously and the suspension interval is reset to the initial length.

The initial short suspension interval seeks to react rapidly to the incorrect detection of contention due to fluctuations in *BGShare*. By checking it again shortly after the first suspension of background processes, the algorithm tries to minimize periods of unnecessary suspension. The exponential growth of the interval, on the other hand, aims to reduce the interference with foreground activities. It rapidly results in a long suspension interval, soon making background processes executed infrequently.

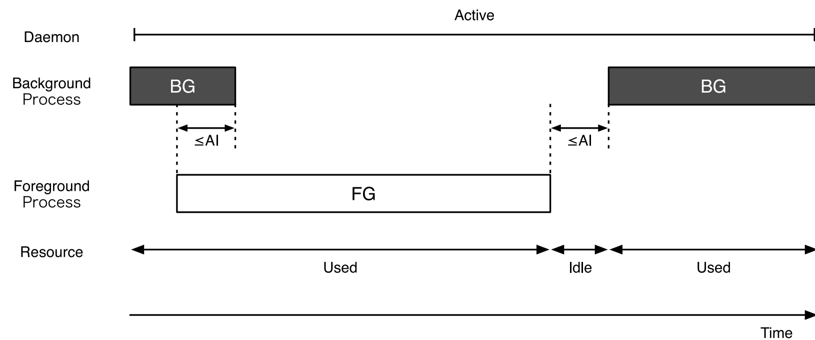
Because the EII algorithm actually executes background processes in order to check their contention with foreground activities, the daemon can precisely know whether it is appropriate to resume those background processes. Another advantage of this algorithm is that the daemon does not need to analyze statistics reported by probes when the background processes are suspended. A disadvantage of the algorithm, on the other hand, is that there exists some time after active foreground work completes during which resources are not efficiently utilized. Only when the current suspension interval has elapsed can background processes be resumed, and thus most resources remain idle until that time.

Our current implementation sets the initial length of background process suspension to 1 second, and the maximum suspension length to 16 seconds.

© 2011 Information Processing Society of Japan



(1) EII Algorithm



(2) IPD Algorithm

Fig. 5 Background execution control. The figure shows how background process execution is controlled by the EII and IPD algorithms. "AI" stands for the aggregation interval of system statistics. Notice the difference between the two algorithms when the daemon is actively analyzing system statistics, and when and how long the resource is idle.

5.2.2 The Idle Period Detection Algorithm

Our second algorithm, called the IPD algorithm, analyzes the statistics of processes other than background processes, instead of actually executing them, to determine when to resume background process execution. The bottom part of Fig. 5 depicts how this algorithm works. During background process suspension, the algorithm seeks a point at which no foreground processes actively consume resources, so that background processes can be executed.

The algorithm uses certain conditions to detect idleness of each resource type. For CPUs, we obtain the approximated CPU share of the swapper process. A high swapper share implies that only a fraction of the CPU capacity is used for foreground processes, and thus background activities may be restarted. We performed a 30-minute trace of CPU statistics in a situation where no active foreground processes exist and only system services are running. During this trace, the approximated swapper share never fell below 62% with the interval for aggregating statistics set to 1 second. We used this value as the threshold to infer CPU idleness. For disks and network interfaces, we simply observe whether any requests to these resources existed during the last statistics aggregation interval. If the swapper resource share is not lower than the threshold and both disks and network interfaces stay idle, the algorithm concludes that background processes can be resumed.

The primary advantage of the IPD algorithm is that it continuously observes system statistics and resumes background processes as soon as it judges resources are idle. With this algorithm, background processes do not suffer unnecessary suspension as in the case of the EII algorithm. On the other hand, a disadvantage of the algorithm is that the daemon always needs to analyze reported statistics as long as background processes exist, whether or not they are suspended. Also, the algorithm is conservative in that it allows resuming background activities only when no requests to disks and network interfaces exist in the system. In practice, this conservative approach works well because our target environments are those in which resources are underutilized and are serving no requests.

5.2.3 Comparison of the Two Algorithms

The EII and IPD algorithms differ from each other in two main aspects: when they observe system statistics and when they execute background processes. This section describes what workloads the two algorithms are expected to handle well, as a result of these differences.

The EII algorithm is intended to perform well when CPUs in the system are busy or fairly loaded, and disk or network contention is not significant. While the IPD algorithm keeps analyzing system statistics throughout the lifetime of back-

© 2011 Information Processing Society of Japan

Table 1 Results of TCP microbenchmarks with a 100% resource intensity. The table shows the increase in foreground execution time for different number of background processes and different background execution control.

	EII	IPD	Low Prio.
1 BG Process	8.5%	0.9%	87.6%
2 BG Processes	11.0%	0.8%	188.3%
4 BG Processes	13.2%	0.8%	388.8%

ground processes, the EII algorithm does not, resulting in lighter CPU resource requirements. However, it needs to execute those background processes when they may compete with foreground processes. The impact of this background process execution on foreground throughput is far less when resource contention is primarily for CPUs than when it is mainly for disks or networks. We present examples where the EII algorithm performs better than the IPD algorithm in our case study with SETI@home, which is shown in Section 6.2.1.

The IPD algorithm, on the other hand, generally works well when foreground and background processes compete severely for disk or network resources. In such a situation, consistent CPU consumption by the algorithm matters little since disk or network performance accounts largely for foreground throughput. Background process execution by the EII algorithm in this case causes disk or network contention, often resulting in a non-negligible impact on the throughput. A good example of this case is our microbenchmarks with intensive network resource consumption, which is explained with **Table 1** in Section 6.1.2.

6. Experiments

We performed experiments to examine the effectiveness of our approach. The experiments described in this section were conducted on the same test machines mentioned in Section 4.4, except for multiprocessor microbenchmarks presented in Section 6.1.3. These measurements were performed on a machine with two 2.33 GHz Dual-Core Intel Xeon processors, with 2 GB of memory and a 250 GB 7200 RPM disk.

6.1 Microbenchmarks

We ran disk and TCP microbenchmarks to show that our daemon appropriately handles different workloads. We used the same programs as the ones for obtaining the sample disk and network statistics, and varied their resource intensity; we selected 100%, 50%, and 12.5%. For different number of background processes, we tried all possible combinations of their resource intensity. The number of foreground processes was fixed to 1, and its resource intensity was also varied. In addition, we ran simple multiprocessor microbenchmarks to show that the enhancement to our basic approach described in Section 4.3.2 guarantees appropriate CPU allocation in multiprocessor environments.

6.1.1 Disk Microbenchmarks

The top row of **Fig. 6** shows the results of our disk microbenchmarks. The top left graph indicates that our system effectively preserves good foreground performance. With the EII algorithm, it keeps the increase in foreground execution time in a range between 6.4% and 12.5%. Because the algorithm executes existing background processes from time to time, foreground performance declines slightly as their number increases. The IPD algorithm outperforms the EII algorithm consistently, sustaining increases of about 1% in foreground execution time across the different number of background processes. Without our system's explicit control over the background processes, foreground performance is severely degraded due to excessive disk seek induced by the multiple running processes.

The top right graph in Fig. 6 shows that our system even improves background execution time. Because it suspends the background processes explicitly while the foreground process exists, their execution time is in general expected to be made longer by our system. However, our system reduces the number of processes running simultaneously, and thus the number of files accessed by these processes, resulting in less disk seek time in total. Comparison of our two algorithms shows that the EII algorithm results in better background execution time as the number of background processes increases. With the IPD algorithm, more disk requests by the background processes remain to be issued when the foreground process completes, and they induce a longer period of inefficient disk seek during which the underlying scheduler keeps switching among the remaining processes. The normalized execution time of 1, 2, or 4 background processes would ideally be 2.0, 3.0, or 5.0, respectively; the foreground process would execute exclusively and then the background processes run concurrently, finishing roughly at the same time. The difference from those numbers in the cases with 2 and 4 background

© 2011 Information Processing Society of Japan

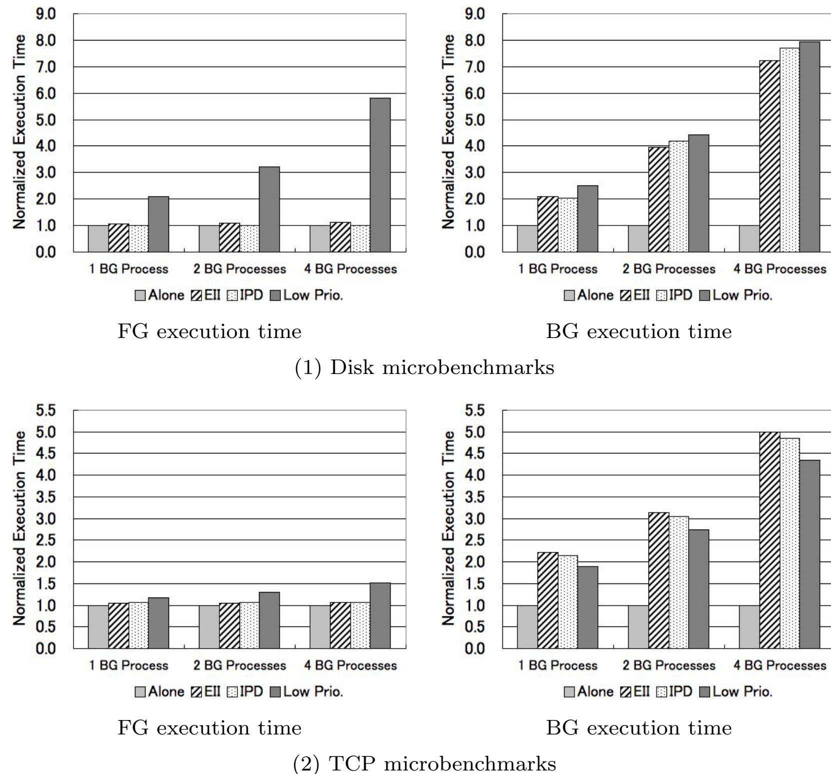


Fig. 6 Results of our disk and TCP microbenchmarks. The figure shows normalized execution times of our disk and TCP microbenchmarks for different number of background processes. The graphs in the top row are disk results and those in the bottom row are TCP results. Bars labeled "Alone" and "Low Prio." indicate the execution time of processes when they are executed alone and when background processes are simply executed with the maximum nice value without explicit control, respectively. In cases with multiple background processes, their average execution time is reported.

processes is due to disk contention.

6.1.2 TCP Microbenchmarks

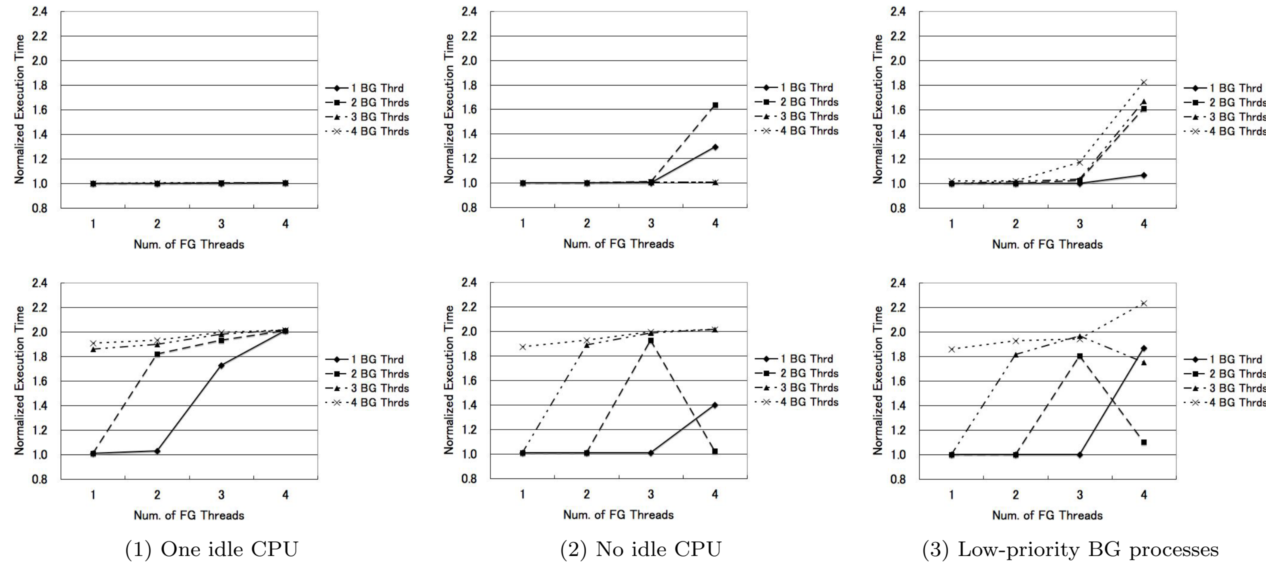
The results of our TCP microbenchmarks are shown in the bottom row of Fig. 6. Compared to disk access, packet transmission requires much less service time. Thus, the interference of the background processes with the foreground process is avoided relatively well by the underlying scheduler, especially when the resource intensity of processes is 50% or 12.5%. As a result, when seen as the average over all of the measurement results as reported in the figure, the benefit of our system in our TCP microbenchmarks is less dramatic than in our disk microbenchmarks. Still, our system reduces the increase in foreground execution time by over 10% when the number of background processes is 1, and by over 45% when their number is 4. At a closer look, the EII algorithm keeps the increase between 3.9% and 5.7%, and the IPD algorithm between 5.8% and 6.1%. Also, because of the relatively light effect of contention, the background execution time with 1, 2, or 4 background processes approximately follows the expected value of 2.0, 3.0, or 5.0, respectively. Note that in some cases the actual execution time is shorter than the expected value, because concurrent execution in these cases leads to efficient utilization of resources with multiple processes available for exploiting them.

When TCP microbenchmark processes send packets intensively, their mutual interference is far greater than the averaged results shown in Fig. 6. Table 1 shows a subset of the TCP microbenchmark results where the resource intensity of all processes is 100%. Without our system, the foreground process incurs significant increases in its execution time. Our system, in this case, provides remarkable performance improvements.

6.1.3 Multiprocessor Microbenchmarks

In order to show that the improvement technique described in Section 4.3.2 handles multiprocessor allocation appropriately, we implemented and tested a prototype that uses an improved version of the IPD algorithm. As described in that section, the enhanced algorithm basically analyzes *BGShare* per CPU, and suspends those background processes that are consuming a contended CPU. It guarantees that at least one CPU is idle when background processes are executed. Also, it judges if there exists idle CPUs and, if so, launches a set of background processes whose total number of active threads does not exceed the number of available CPUs. The algorithm resumes all outstanding background processes if no CPUs are actively used by foreground processes. The prototype uses an interval of 0.25 seconds for aggregating system statistics. Since the aim is to show the capability to handle simple multiprocessor cases with CPU-intensive

© 2011 Information Processing Society of Japan



(1) One idle CPU (2) No idle CPU (3) Low-priority BG processes

Fig. 7 Results of multiprocessor microbenchmarks. The figure shows execution time of foreground and background processes in a multiprocessor environment for three different cases: (1) the system ensures at least one CPU is idle while executing background processes, (2) the system considers each CPU independently without keeping a CPU idle during background process execution, and (3) background processes are executed simply with low priorities. The top and bottom rows show foreground and background performance, respectively. In each graph, the x-axis indicates the number of active foreground threads and the y-axis shows normalized execution time of processes.

processes, this short interval suffices for our purposes.

In this experiment, we used a simple benchmark program whose threads consume a certain amount of CPU time in loops. We ran this program in three different cases: (1) the system ensures at least one CPU is idle while executing background processes, (2) the system considers each CPU independently without keeping a CPU idle during background process execution, and (3) background processes are executed simply with low priorities. In each case, we executed one foreground and one background processes, varying the number of threads they launch, and measured their execution time. The foreground process started execution 10 seconds after the start of the background process.

The results are summarized in **Fig. 7**. The leftmost column of the figure shows that our prototype strictly prioritizes the foreground process and preserves foreground performance in all cases. When the total number of foreground and background threads equals or exceeds 4, it suspends the background process and therefore its execution time stays close to 2.0. The middle column shows the case in which our prototype considers each CPU separately and does not guarantee that one or more CPUs are idle when executing the background process. The results indicate that when the number of foreground threads is 3 or 4, the underlying scheduler can choose to prioritize the foreground threads by using more CPUs for them than for the background threads. As a result, foreground execution time grows and the corresponding background execution time drops. (Notice when the number of foreground threads is 3, background execution time can be below 2.0 despite all the CPUs being utilized, slightly affecting foreground execution time.) Note that the scheduler's CPU allocation pattern varied across

© 2011 Information Processing Society of Japan

the measurements. Even with the same number of foreground and background threads, the scheduler may decide to differentiate the number of CPUs assigned to foreground and background threads or to allocate a fraction of each CPU's capacity to background threads. Finally, the rightmost column of the figure shows the case in which the background process is executed without control by our prototype. The performance of the foreground process in this case is still worse than the case of the middle column. Foreground execution time is remarkably increased when the number of foreground threads is 4.

6.2 Case Studies

To examine the effectiveness of our system in practical situations, we conducted experiments with three kinds of background applications: scientific computing, disk error checking, and network file transfer.

6.2.1 Scientific Computing

In our first case study, we executed SETI@home as a background activity and measured the execution time of different foreground programs. As the foreground processes, we ran `fftw-wisdom` [13], `make`, and `pcregrep`. `fftw-wisdom` is a CPU-intensive program that generates information regarding optimal computation of the Fourier transform. We used `make` to perform compilation of Apache 2.2.2 and `pcregrep`, a variant of grep, to search for a certain word under a directory containing Linux 2.6.16 source code. The file system containing the Apache and Linux source code was remounted before each measurement to clear cached data. Also, because the priority of a CPU-intensive process can change considerably on Solaris 10, we rebooted the test machine before each measurement of `fftw-wisdom`'s execution time in order to obtain results in a steady condition. We report only the results of foreground performance for this case study, as we were not able to measure the execution time of a reproducible computation using SETI@home.

As shown in **Fig. 8**, a CPU-intensive process with a low priority like SETI@home is kept from interfering with other processes strictly on Solaris 10. Still, when the foreground process is `fftw-wisdom`, which is also CPU-intensive, its execution time is increased by over 12%. We attribute this increase to the fact that the priority of `fftw-wisdom` kept decreasing during its execution and became lower than other processes, approaching that of SETI@home. This phenomenon

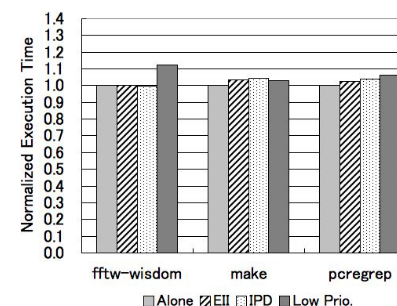


Fig. 8 Results of background SETI@home measurements. The figure shows foreground execution time of `fftw-wisdom`, `make`, and `pcregrep` running with SETI@home in the background.

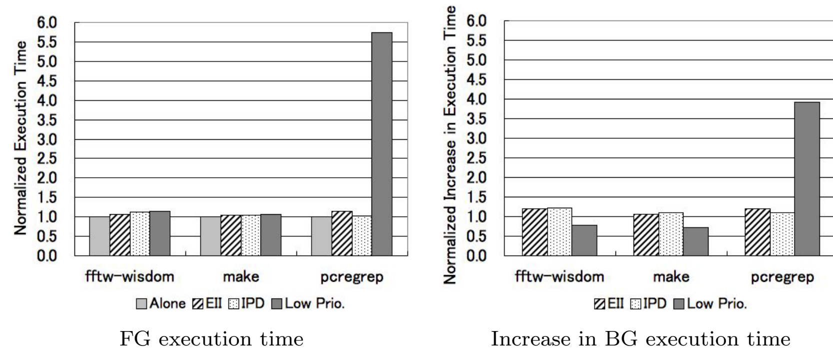
indicates that the performance of a CPU-intensive process can be affected considerably by other low-priority processes even if it is initially assigned a default priority. On the other hand, both of our algorithms preserve good performance of `fftw-wisdom` by suspending SETI@home, and hardly induce any increase on its execution time.

When the foreground process is `make` or `pcregrep`, the underlying scheduler preserves fairly good foreground execution time. Because `make`, including its child processes, and `pcregrep` issue disk requests, their priorities tend to stay high and thus their execution time incurs only a small increase even without our system. In addition, `make` requires our system to process more information reported by probes, such as the creation and completion of processes, than the other two foreground programs. Our system thus does not improve the foreground performance. Still, the increases in foreground execution time with our two algorithms differ from the increase in the low-priority case, which is 3.1%, by only 1.2% or less. In the `pcregrep` case, our system slightly improves the foreground execution time of the low-priority case, lowering the increase from 6.3% to 2.4% and to 3.7% using the EII and IPD algorithms, respectively.

6.2.2 Disk Error Checking

Our second study is disk error checking with fsck. We executed fsck as a background process, and the same foreground processes we used for the case study with SETI@home. The files accessed by the foreground processes and

© 2011 Information Processing Society of Japan



FG execution time   Increase in BG execution time

Fig. 9 Results of background fsck measurements. The figure shows foreground execution time of `fftw-wisdom`, `make`, and `pcregrep` running with fsck in the background, and the corresponding increases in fsck's execution time.

disk slices checked by fsck reside on the same disk, resulting in contention when accessed simultaneously.

The results of the measurements are shown **Fig. 9**. The left graph in the figure shows the execution time of the three foreground programs, and the right graph shows the increases in fsck's execution time normalized by the execution time of the foreground processes. Notice that we report the increases in the right graph, not the execution time itself, as the execution time of the foreground processes is not directly comparable to that of fsck.

When the foreground process is `fftw-wisdom` or `make`, the impact of resource contention is fairly small and so is foreground performance degradation. Still, `fftw-wisdom` incurs a 14.6% increase in its execution time without background execution control. The EII and IPD algorithms reduce the increase to 7.1% and 11.9%, respectively. As in our SETI@home case study, we observed dynamic decreases in the priority of `fftw-wisdom` resulting in fluctuations in its execution time. These fluctuations caused increases in the average execution time reported in Fig. 9. When the foreground process is `make`, the original increase in its execution time is 6.1% and our two algorithms improve it slightly.

Unlike `fftw-wisdom` and `make`, `pcregrep` competes significantly for disk access with fsck, suffering severe performance degradation. Without our background execution control, fsck causes the execution time of `pcregrep` to be almost 6 times longer than its original execution time. Our system significantly improves foreground execution time for this combination of foreground and background processes, and reduces the increase in execution time to 13.5% with the EII algorithm and to 3.3% with the IPD algorithm.

The right graph in Fig. 9 indicates that our system keeps the suspension time of background processes close to the foreground execution time. When the foreground process is `fftw-wisdom` or `make`, in which case resource contention is moderate, the increase in background execution time is slightly smaller with the EII algorithm than with the IPD algorithm. The EII algorithm executes background fsck from time to time, and thus it makes small progress. This progress outweighs the benefit of using the IPD algorithm, which avoids periods of excess background process suspension present with the EII algorithm. In the low-priority cases, the scheduler does not strictly suspend background fsck, and the normalized increases in its execution time are less than 1.0. Note that in these cases, total resource utilization is more efficient than in the EII and IPD cases; the background process completes earlier while foreground performance degradation is insignificant because of the moderate level of contention. Our system conservatively suspends background process execution, eliminating the possibility of sacrificing foreground performance. Therefore, in such cases low-priority execution of background processes may yield higher efficiency of resource use. When the foreground process is `pcregrep`, the IPD algorithm results in slightly more efficient execution of fsck than the EII algorithm since the foreground and background processes suffer significant resource contention when executed simultaneously. Without explicit background execution control, this contention leads to a large increase in fsck's execution time, as well as `pcregrep`'s execution time.

6.2.3 Network File Transfer

Our third case study examines the effectiveness of our system in controlling background network data transmission. As foreground and background processes, we executed scp that copied the same Linux 2.6.16 source directory on a source node to different locations on a destination node. Because both foreground and background processes access the same files on the source node, most of the time only the first process accessing the files reads them from the disk and the other processes read cached data. Thus, there exists little disk contention at the source

© 2011 Information Processing Society of Japan

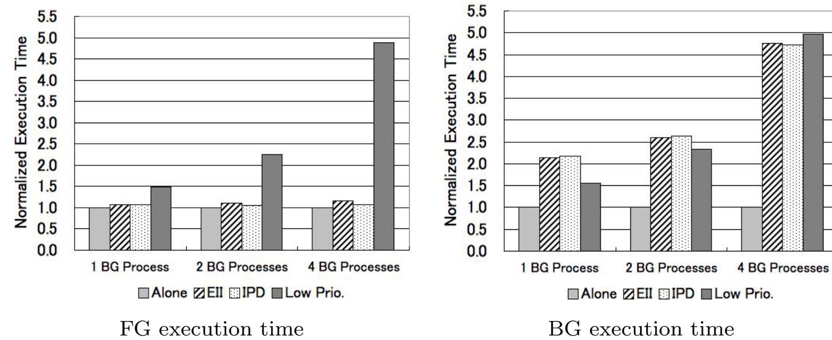


Fig. 10 Results of scp measurements. The figure shows normalized execution time of foreground and background scp for different number of background scp processes.

node. In addition, the source directory was accessed before measurements were performed so that part of the data resided in the cache on the source node.

Figure 10 summarizes the results. When background scp processes are executed simply with low priorities, without external control, they cause significant degradation of foreground scp throughput. This degradation deteriorates as the number of background processes increases. With 4 background scp's, foreground execution time grows to as much as 489%. Even with only 1 background scp, the execution time rises to 148%. Also, the fact that the foreground execution time in these cases is similar to the corresponding background execution time emphasizes the ineffective prioritization of processes by the underlying scheduler. There is a good chance that these overheads discourage users from actively performing background data backup or replication over a network.

With the EII algorithm, the increases in foreground scp's execution time are kept between 6.9% and 16.9%. As observed in other experiments, the foreground execution time grows moderately as the number of background processes increases. With the IPD algorithm, foreground scp incurs less interference of the background processes and the increases in its execution time are kept between 5.3% and 6.7%. Without explicit control of background processes, their execution time grows as their number increases, along with the foreground execution time. When there are 4 of them, the average execution time exceeds that of the EII and IPD cases. Compared to the other two cases, there are more processes in the low-priority case that compete simultaneously for network data transmission on the source node and for disk I/O on the destination node.

7. Discussion

In this section, we discuss essential issues about our approach. We first describe how our system affects the response time of foreground processes. We then examine the necessity of adjusting thresholds for detecting resource contention for different system environments, and discuss the impact of using our system on the progress of background network activity.

7.1 Response Time of Foreground Processes

As mentioned in Section 1, our approach focuses on improving overall system throughput without foreground throughput degradation, and does not address the response time of foreground processes. As a result, the response time is affected if background processes are running at the time when foreground processes start execution. Specifically, for applications whose response time is shorter than the statistics aggregation interval, their foreground response time would be the same whether or not our system is used to control background processes. Because our implementation checks aggregated resource statistics every 0.25 seconds, in some cases background processes may be suspended sooner than a whole aggregation interval elapses after the emergence of resource contention. As soon as *BGShare* of any resource decreases below the corresponding threshold, those background processes are stopped. Still, if foreground response time is well shorter than the aggregation interval, there would be little benefit.

Foreground applications with longer response times are affected differently by our two algorithms for background execution control. With the EII algorithm, the increase in the response time depends on how many periods of temporary background execution exist during foreground processing. Because the suspension interval is short at the beginning of foreground process execution, the increase may be non-negligible. With the IPD algorithm, the duration of interference between foreground and background processes is expected to be no longer than the statistics aggregation interval. With the cost of probes ignored, only this interference affects foreground response time, as the background processes are

© 2011 Information Processing Society of Japan

suspended completely until the completion of foreground execution.

The fact that our approach does affect foreground response time is due to its reactive nature. We primarily seek to help improve overall system throughput while keeping foreground throughput unaffected. We thus would like to exploit whatever resources are idle. Only when background processes cause resource contention and their suspension does not result in wasted resources, do we actually suspend them. On the other hand, if we were to avoid degrading foreground response time, we would need to ensure that resources are immediately and fully available to foreground processes when they request those resources. Achieving this would generally require keeping resources idle in advance when resource requests by foreground processes are expected. However, doing so is against our objective of keeping resources as busy as possible.

7.2 Adapting to Diverse System Configurations

We expect that our method of detecting resource contention based on thresholds can be applied easily and effectively to systems with diverse workload characteristics. As shown in Fig. 2, our system can detect resource contention accurately with a wide range of thresholds for each resource type, and the efficacy of our approach is thus insensitive to the actual values of these thresholds. Consequently, although a certain set of threshold values may not be ideal for any possible workloads, our system would work as mostly intended under diverse workloads without being specifically customized. Such easiness and accuracy in detecting resource contention make our approach general and broaden its applicability.

Furthermore, estimating appropriate values for the thresholds is straightforward in general as our approach is based on the resource usage of background processes relative to that of foreground processes. For the disk workloads shown in Fig. 2, for instance, when there are one foreground and one background processes, the accuracy of detecting resource contention drops around the disk threshold of 50%. When there are two or four background processes, the accuracy decreases with the threshold of approximately 66% and lower, or 80% and lower, respectively. These points at which the accuracy starts to drop reflect the number of background processes divided by that of both foreground and background processes. Also, because CPU and network resource shares of background processes decrease more dramatically than their disk resource share under the existence of resource contention, estimating proper thresholds over these shares would be even easier. If we used a metric for inferring resource contention based on some absolute value about resource usage, such as the number of bytes transferred to and from disks by background processes, the value of such a metric corresponding to resource contention would be application- and workload-specific.

For these reasons, it would not be crucial to customize the thresholds upon installation of our system in order for it to detect resource contention. Still, if necessary, determining appropriate thresholds in an automated way would be a simple process. We could execute a set of processes with varied resource needs, such as a subset of those described in Section 4.4.1, and decide a proper range of thresholds over each resource type based on gathered system statistics. As the set of measurements shown in Section 4.4 was intended to be exhaustive, simplifying the calibration process for determining thresholds would be an interesting area for our future work.

7.3 TCP Timeout

If our system suspends a background process with a TCP connection for a long period of time, that connection's timeout may occur. If such timeout happens frequently, it will lead to lowered background throughput and inefficient use of idle resources. With the EII algorithm for background execution control, this problem is avoided by limiting the maximum length of the suspension interval. Since our current implementation of the algorithm uses the maximum interval of 16 seconds and background processes are run temporarily after each interval, it practically avoids TCP timeout. On the other hand, timeout is more likely with the IPD algorithm. It suspends background processes completely for the duration of foreground process execution for the sake of less interference between foreground and background processes than that caused by the EII algorithm. Therefore, TCP timeout occurs if a foreground process continues running longer than the timeout limit. One solution would be adding a simple enhancement to the algorithm that allows occasional execution of background processes to avoid this problem.

8. Conclusion

In this paper, we proposed an effective user-level approach to controlling back-

© 2011 Information Processing Society of Japan

ground processes for idle resource utilization. We showed that we can reasonably infer the interference of background processes with foreground processes from outside the underlying operating system, by using system statistics readily available at the user level. We obtain approximated resource shares of background processes derived from such statistics, and suspend them when these shares become low as a result of resource contention. Our system implemented on Solaris 10 appropriately suspends background processes and avoids the throughput degradation of foreground processes.

Our approach is characterized by the following advantages. First, it requires no considerable modification to existing systems, encouraging users to actively make use of idle resources. We believe this aspect of our approach is particularly beneficial, given the increasing value of idle resource utilization. Next, our method reflects actual resource usage, and takes different resources into account in examining the existence of resource contention. Finally, it can manage multiple background processes simultaneously. This property allows aggressive exploitation of idle resources by executing background processes with different resource needs.

In obtaining statistics indicative of resource usage, we take advantage of dynamically enabled probes. Probes are a promising approach to exposing valuable system information at the user level, and are becoming widely accepted. As there exists dynamic probes available on common platforms besides Solaris 10, we believe that our method of controlling background processes can be easily applied to them as well.

References

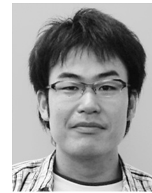
1) Abe, Y., Yamada, H. and Kono, K.: Enforcing Appropriate Process Execution for Exploiting Idle Resources from Outside Operating Systems, *Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (*EuroSys '08*), pp.27–40 (2008).
2) Anderson, D.P.: BOINC: A System for Public-Resource Computing and Storage, *Proc. 5th IEEE/ACM International Workshop on Grid Computing* (2004).
3) Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M. and Werthimer, D.: SETI@home: An Experiment in Public-Resource Computing, *Comm. ACM*, Vol.45, No.11, pp.56–61 (2002).
4) Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Burnett, N.C., Denehy, T.E., Engle, T.J., Gunawi, H.S., Nugent, J.A. and Popovici, F.I.: Transforming Policies into Mechanisms with Infokernel, *Proc. 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp.90–105 (2003).
5) Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M.E., Becker, D., Chambers, C. and Eggers, S.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating Systems Principles* (*SOSP '95*), pp.267–283 (1995).
6) Brakmo, L.S., O'Malley, S.W. and Peterson, L.L.: TCP Vegas: New Techniques for Congestion Detection and Avoidance, *Proc. ACM SIGCOMM '94*, pp.24–35 (1994).
7) Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H.: Dynamic Instrumentation of Production Systems, *Proc. USENIX 2004 Annual Technical Conference* (*USENIX '04*), pp.15–28 (2004).
8) Douceur, J.R. and Bolosky, W.J.: Progress-based regulation of low-importance processes, *Proc. 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pp.247–260 (1999).
9) DTrace - FreeBSD Wiki. http://wiki.freebsd.org/DTrace
10) dtrace(1) Mac OS X Manual Page. http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man1/dtrace.1.html
11) Eggert, L. and Touch, J.D.: Idletime Scheduling with Preemption Intervals, *Proc. 20th ACM Symposium on Operating Systems Principles* (*SOSP '05*), pp.249–262 (2005).
12) Engler, D.R., Kaashoek, M.F. and James O'Toole, J.: Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. 15th ACM Symposium on Operating Systems Principles* (*SOSP '95*), pp.251–266 (1995).
13) FFTW Home Page. http://www.fftw.org
14) Golding, R., Bosch, P., Staelin, C., Sullivan, T. and Wilkes, J.: Idleness is not sloth, *Proc. USENIX 1995 Technical Conference*, pp.201–212 (1995).
15) Kureger, P. and Chawla, R.: The Stealth Distributed Scheduler, *Proc. 11th International Conference on Distributed Computing Systems* (*ICDCS '91*), pp.336–343 (1991).
16) Kuzmanovic, A. and Knightly, E.W.: TCP-LP: A Distributed Algorithm for Low Priority Data Transfer, *Proc. IEEE INFOCOM*, pp.1691–1701 (2003).
17) Larson, S.M., Snow, C.D., Shirts, M. and Pande, V.S.: Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology, *Computational Genomics* (2002).
18) Linux Technology Center: Welcome. http://sourceware.org/systemtap/kprobes
19) Litzkow, M.J., Livny, M. and Mutka, M.W.: Condor - A Hunter of Idle Workstations, *Proc. 8th International Conference on Distributed Computing Systems* (*ICDCS '88*), pp.104–111 (1988).
20) Lumb, C.R., Schindler, J. and Ganger, G.R.: Freeblock Scheduling Outside of Disk Firmware, *Proc. 1st USENIX Conference on File and Storage Technologies* (*FAST*

© 2011 Information Processing Society of Japan

'02), pp.275–288 (2002).
21) Lumb, C.R., Schindler, J., Ganger, G.R., Nagle, D.F. and Riedel, E.: Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives, *Proc. 4th Symposium on Operating Systems Design and Implementation* (*OSDI '00*), pp.87–102 (2000).
22) Moore, R.J.: A Universal Dynamic Trace for Linux and Other Operating Systems, *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference* (*USENIX '01*), pp.297–308 (2001).
23) Tamches, A. and Miller, B.P.: Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, *Proc. 3rd Symposium on Operating Systems Design and Implementation* (*OSDI '99*), pp.117–130 (1999).
24) Venkataramani, A., Kokku, R. and Dahlin, M.: TCP Nice: A Mechanism for Background Transfers, *Proc. 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), pp.329–344 (2002).

(Received July 26, 2010)
(Accepted November 12, 2010)

Yoshihisa Abe is a Ph.D. student in the Computer Science Department at Carnegie Mellon University. He received his M.S. degree in Computer Science from New York University in 2009, and his B.E. degree in Information and Computer Science from Keio University in 2007. His research interests lie in the areas of operating systems, virtualization, and resource management. He is a member of ACM.



Hiroshi Yamada was born in 1981. He received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently a research associate of the Faculty of Science and Technology at Keio University. His research interests include operating systems, virtualization, and system software. He is a member of ACM, USENIX and IEEE/CS.

Kenji Kono received his B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in Computer Science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, dependable systems, and Internet security. He is a member of IEEE/CS, ACM and USENIX.

© 2011 Information Processing Society of Japan