A Hierarchical CPU Scheduler for Multimedia Operating Systems*

Pawan Goyal, Xingang Guo, and Harrick M. Vin

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188
E-mail: {pawang,xguo,vin}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885
URL: http://www.cs.utexas.edu/users/dmcl

Abstract

The need for supporting variety of hard and soft real-time, as well as best effort applications in a multimedia computing environment requires an operating system framework that: (1) enables different schedulers to be employed for different application classes, and (2) provides protection between the various classes of applications. We argue that these objectives can be achieved by *hierarchical partitioning* of CPU bandwidth, in which an operating system partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. We present Start-time Fair Queuing (SFQ) algorithm, which enables such hierarchical partitioning. We have implemented a hierarchical scheduler in Solaris 2.4. We describe our implementation, and demonstrate its suitability for multimedia operating systems.

1 Introduction

Over the past few years, computing, communication, and video compression technologies have advanced significantly. Their synergistic advances have made the bandwidth and the storage space requirements of digital video manageable, and thereby have enabled a large class of multimedia applications (e.g., video conferencing, distance learning, news-on-demand services, virtual reality simulation of fire fighting, etc.). Since digital audio and video convey appropriate meaning only when presented continuously in time, such applications impose real-time requirements on the underlying storage, transmission, and processor sub-systems. Specifically, they require an operating system to allocate resources such as CPU, I/O bus, disk, and network bandwidth in a predictable manner as well as provide Quality of Service (QoS) guarantees (in terms of throughput, response time, etc.). Since no existing operating system meets these requirements, realizing such applications requires conventional operating systems to be extended along several dimensions. Design and implementation of a CPU allocation framework suitable for multimedia operating systems is the subject matter of this paper.

To determine suitable CPU scheduling algorithms, consider the requirements imposed by various application classes that may co-exist in a multimedia system:

- *Hard real-time applications*: These applications require an operating system to deterministically guarantee the delay that may be experienced by various tasks. Conventional schedulers such as the Earliest Deadline First (EDF) and the Rate Monotonic Algorithm (RMA) are suitable for such applications [12].

- *Soft real-time applications*: These applications require an operating system to statistically guarantee QoS parameters such as maximum delay and throughput. Since a large number of such applications are expected to involve video, consider the processing requirements for variable bit rate (VBR) video:

  Due to inherent variations in scene complexity as well as the use of intra- and inter-frame compression techniques, processing bandwidth required for compression and decompression of frames of VBR video varies highly at multiple time-scales. For instance, Figure 1 illustrates that the processing bandwidth required for decompressing MPEG video varies from frame-to-frame (i.e., at the time scale of tens of milliseconds) as well as from scene-to-scene (i.e., at the time scale of seconds). Furthermore, these vari-

*This research was supported in part by IBM Graduate Fellowship, IBM Faculty Development Award, Intel, the National Science Foundation (Research Initiation Award CCR-9409666), NASA, Mitsubishi Electric Research Laboratories (MERL), and Sun Microsystems Inc.

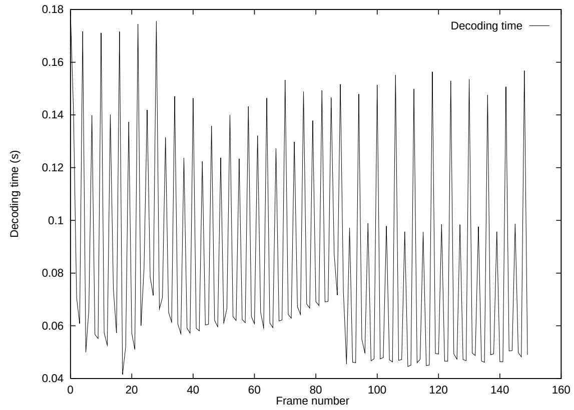

Figure 1 : Variation in decompression times of frames in an MPEG compressed video sequence

ations are unpredictable. These features lead to the following requirements for a scheduling algorithm for VBR video applications:

- Due to the multiple time-scale variations in the computation requirement of video applications, to efficiently utilize CPU, an operating system will be required to over-book CPU bandwidth. Since such over-booking may lead to CPU overload (i.e., cumulative requirement may exceed the processing capacity), a scheduling algorithm must provide some QoS guarantees even in the presence of overload.
- Due to the difficulty in predicting the computation requirements of VBR video applications, a scheduling algorithm must not assume precise knowledge of computation requirements of tasks.

EDF and RMA schedulers do not provide any QoS guarantee when CPU bandwidth is overbooked. Furthermore, their analysis requires the release time, the period, and the computation requirement of each task (thread) to be known a priori. Consequently, although appropriate for hard real-time applications, these algorithms are not suitable for soft real-time multimedia applications. Hence, a new scheduling algorithm that addresses these limitations is desirable.

- *Best-effort applications*: Many conventional applications do not need performance guarantees, but require the CPU to be allocated such that average response time is low while the throughput achieved is high. This is achieved in current systems by time-sharing scheduling algorithms.

From this, we conclude that different scheduling algorithms are suitable for different application classes in a multimedia system. Hence, an operating system framework that enables different schedulers to be employed for different applications is required. In addition to facilitating co-existence, such a framework should provide protection between the various classes of applications. For example, it should ensure that the overbooking of CPU for soft real-time applications does not violate the guarantees of hard real-time applications. Similarly, misbehavior of soft/hard real-time applications, either intentional or due to a programming error, should not lead to starvation of best-effort applications.

The requirements for supporting different scheduling algorithms for different applications as well as protecting application classes from one another leads naturally to the need for *hierarchical partitioning* of CPU bandwidth. Specifically, an operating system should be able to partition the CPU bandwidth among various application classes, and each application class, in turn, should be able to partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. In this paper, we present a flexible framework that achieves this objective.

In our framework, the hierarchical partitioning is specified by a tree. Each thread in the system belongs to exactly one leaf node, and each node in the tree represents either an application class or an aggregation of application classes. Whereas threads are scheduled by leaf node dependent schedulers (determined by the requirements of the application class), intermediate nodes are scheduled by an algorithm that achieves hierarchical partitioning. Specifically, intermediate nodes must be scheduled by an algorithm that: (1) achieves fair distribution[1] of processor bandwidth among competing nodes, (2) does not require a priori knowledge of computational requirements of threads, (3) provides throughput guarantees, and (4) is computationally efficient. We present Start-time Fair Queuing (SFQ) algorithm which meets all of these requirements. We further demonstrate that SFQ is suitable for video applications. We have implemented our hierarchical scheduling framework in Solaris 2.4. We describe our implementation and evaluate its performance. Our results demonstrate that the framework: (1) enables co-existence of heterogeneous schedulers, (2) protects application classes from each other, and (3) does not impose higher overhead than conventional time-sharing schedulers.

Observe that our hierarchical partitioning framework

[1]Intuitively, a CPU allocation is fair if, in every time interval, all runnable threads receive the same fraction of CPU bandwidth. This notion of uniform fairness generalizes to weighted fairness when threads have different weights and each thread receives CPU bandwidth in proportion to its weight. We will formalize this notion in Section 3.

also facilitates the development of a QoS manager that allocates resources as per the requirements of applications [10]. To illustrate, if an application requests hard (soft) real-time service, then the QoS manager can use a deterministic (statistical) admission control algorithm which utilizes the capacity allocated to hard (soft) real-time classes to determine if the request can be satisfied, and if so, assign it to the appropriate partition. On the other hand, if an application requests best-effort service, then the QoS manager would not deny the request but assign it to an appropriate partition depending on some other resource sharing policies. A QoS manager can also dynamically change the hierarchical partitioning to reflect the relative importance of various applications. For example, initially soft real-time applications may be allocated very small fraction of the CPU, but when many video decoders requesting soft real-time services are started (possibly as a part of a video conference), the allocation of soft real-time class may be increased significantly. The development of such policies, however, is the subject of future research and beyond the scope of this paper.

The rest of this paper is organized as follows. In Section 2, we introduce our hierarchical CPU scheduling framework. The Start-time Fair Queuing (SFQ) scheduling algorithm and its properties are described in Section 3. The details of our hierarchical SFQ scheduler implementation are described in Section 4. Section 5 describes the results of our experiments. We present related work in Section 6, and summarize our results in Section 7.

2 A Framework for Hierarchical CPU Scheduling

In our framework, the hierarchical partitioning requirements are specified through a *tree* structure. Each thread in the system belongs to exactly one leaf node. Each leaf node represents an aggregation of threads[2], and hence an application class, in the system. Each non-leaf node in the tree represents an aggregation of application classes. Each node in the tree has a weight that determines the percentage of its parent node's bandwidth that should be allocated to it. Specifically, if $r_1, r_2, ..., r_n$ denote the weights of the n children of a node, and if B denotes the processor bandwidth allocated to the parent node, then the bandwidth received by node i is given by:

$$B_i = \left(\frac{r_i}{\sum_{j=1}^{n} r_j} \right) * B$$

Furthermore, each node has a scheduler. Whereas the scheduler of a leaf node schedules all the threads that belong to the leaf node, the scheduler of an intermediate node schedules its child nodes. Given such a scheduling structure, the scheduling of threads occurs hierarchically: the root node schedules one of its child nodes; the child node, in turn, schedules one of its child nodes until a leaf node schedules a thread for execution. Figure 2 illustrates one such scheduling structure. In this example, the root class (node and class are used interchangeably) has three sub-classes: hard real-time, soft real-time and best-effort, with weights 1, 3, and 6, respectively. The bandwidth of the best-effort class has been further divided equally among leaf classes user1 and user2. Furthermore, whereas the soft real-time and user1 leaf classes employ a scheduler that fairly distributes its CPU allocation among its threads, the hard real-time and user2 classes have EDF and time-sharing schedulers, respectively.

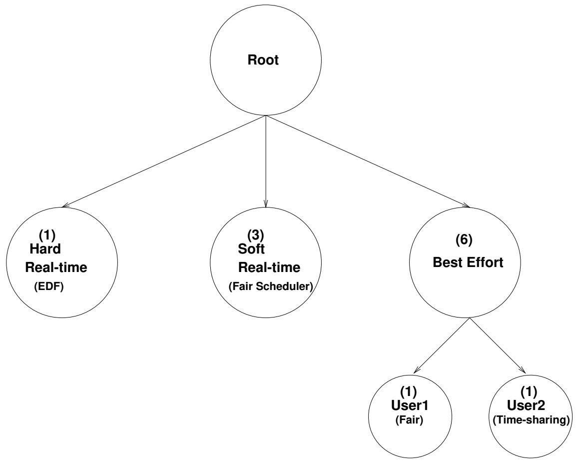

Figure 2 : An example scheduling structure

Observe that the schedulers at leaf nodes in the hierarchy are determined based on the requirements of the applications. The requirements of a scheduling algorithm for intermediate nodes in the hierarchy, on the other hand, can be defined as follows:

1. To achieve hierarchical partitioning, the algorithm for scheduling intermediate nodes in the hierarchy should:
    - Partition the bandwidth allocated to a class among its sub-classes such that each sub-class gets its specified share.
    - Allocate the residual bandwidth fairly among its sub-classes. For example, in Figure 2, if there are no eligible threads in the hard real-time class, then its allocation should be partitioned between the soft real-time and best-effort nodes in the ratio 3:6.

Both these requirements would be met if the schedul-

[2] Threads are assumed to be the scheduling entities in the system.

ing algorithm partitions the allocation of a class among its sub-classes in proportion to their weights, i.e., achieves weighted fairness. Moreover, as the following example illustrates, a key requirement for such an algorithm is that it should achieve weighted fairness even when the bandwidth available to a class fluctuates over time.

Example 1 *Consider the scheduling structure shown in Figure 2. Initially, let there be no threads in the hard and soft real-time classes. Consequently, the best-effort class receives the full CPU bandwidth. When threads join the hard and soft real-time classes, the bandwidth available to the best-effort class goes down to 60% of the CPU bandwidth. In such a scenario, to ensure that user1 and user2 continue to receive equal share of the available bandwidth, the scheduling algorithm for the best-effort class must remain fair even when the available bandwidth fluctuates over time.*

2. Since the computational requirements of tasks may not be known precisely, the scheduling algorithm should not assume a priori knowledge of the time duration for which a task executes before it is blocked.
3. To support hard and soft real-time application classes, the scheduling algorithm should provide bounds on minimum throughput and maximum delay observed by nodes. Furthermore, for the bounds to be useful, they should hold in realistic computing environments in which interrupts may be processed at the highest priority [9].
4. To be feasible in general purpose operating systems, the scheduling algorithm should be computationally efficient.

Recently, we have developed a packet scheduling algorithm, referred to as *Start-time Fair Queuing* (SFQ), which achieves fair allocation of network bandwidth [6]. In the next section, we present the algorithm and demonstrate that it meets the above requirements, and hence, is suitable for CPU scheduling in multimedia operating systems.

3 Start-time Fair Queuing

Start-time Fair Queuing (SFQ) is a resource allocation algorithm that can be used for achieving fair CPU allocation. Before we present SFQ, let us formalize the notion of fair allocation. Let r_f be the weight of thread f and $W_f(t_1, t_2)$ be the aggregate work done in interval $[t_1, t_2]$ by the CPU for thread f. For ease of exposition, let the work done by the CPU for a thread be measured by the number of instructions executed for the thread. Then, a CPU allocation is considered to be fair if, for all intervals $[t_1, t_2]$ in which two threads f and m are runnable, the normalized work (by weight) received by them is identical (i.e., $\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$). Clearly, this is an idealized definition of fairness as it assumes that threads can be served in infinitesimally divisible units. Since the threads are scheduled for a quantum at a time, there will be some unfairness. Consequently, the objective of a fair scheduling algorithm is to minimize the resultant unfairness (i.e., ensure that $\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right|$ is as close to 0 as possible)[3].

To achieve this objective, SFQ assigns a start tag to each thread and schedules threads in the increasing order of start tags. To define the start tag, let the threads be scheduled for variable length quantum at a time. Also, let q_f^j and l_f^j denote the j^{th} quantum of thread f and its length[4] (measured in units of instructions), respectively. Let $A(q_f^j)$ denote the time at which the j^{th} quantum is requested. If the thread is making a transition from a blocked mode to runnable mode, then $A(q_f^j)$ is the time at which the transition is made; otherwise it is the time at which its previous quantum finishes. Then SFQ algorithm is defined as follows:

1. When quantum q_f^j is requested by thread f, it is stamped with start tag S_f, computed as:
$$S_f = \max\{v(A(q_f^j)), F_f\} \tag{1}$$
where $v(t)$ is the virtual time at time t and F_f is the finish tag of thread f. F_f is initially 0, and when j^{th} quantum finishes execution it is incremented as:
$$F_f = S_f + \frac{l_f^j}{r_f} \tag{2}$$
where r_f is the weight of thread f.
2. Initially the virtual time is 0. When the CPU is busy, the virtual time at time t, $v(t)$, is defined to be equal to the start tag of the thread in service at time t. On the other hand, when the CPU is idle, $v(t)$ is set to the maximum of finish tag assigned to any thread.
3. Threads are serviced in the increasing order of the start tags; ties are broken arbitrarily.

---

[3] Several other definitions of fairness have been introduced in the networking and operating systems literature. A comparative evaluation of their relative merits, however, is beyond the scope of this paper.

[4] If l_f^j is the length of quantum q_f^j in terms of instructions, then its time duration is $t_f^j = \frac{l_f^j}{C}$ where C is the rate of execution of the CPU.

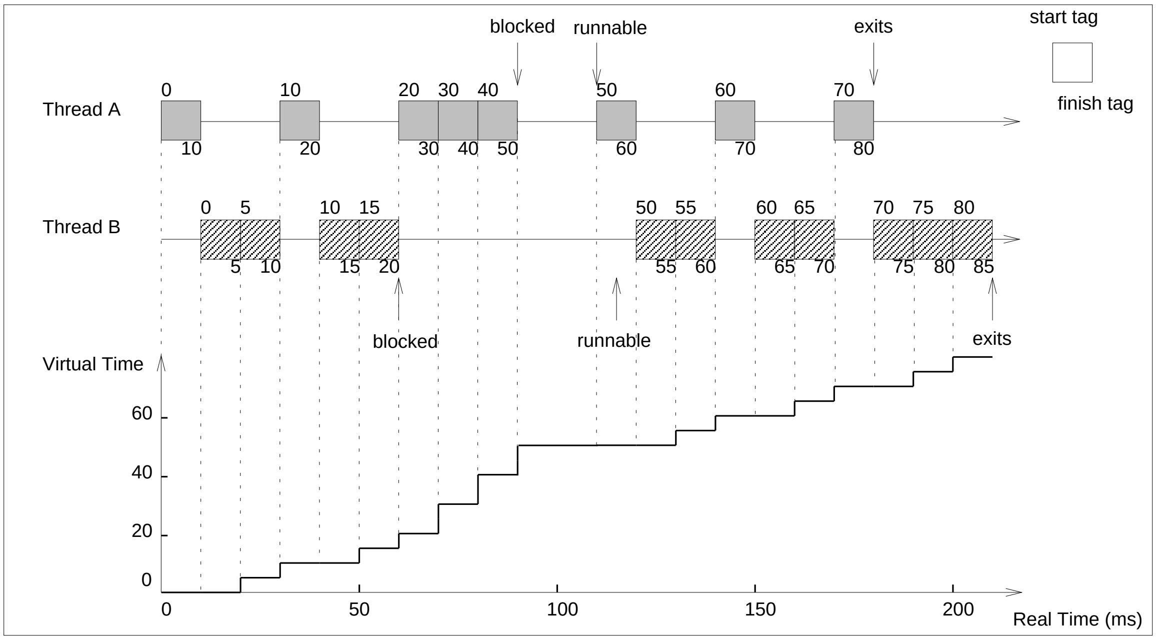

Figure 3 : Computation of virtual time, start tag, and finish tag in SFQ: an example

The following example illustrates the computation of the virtual time, as well as the start and the finish tags (and hence, the process of determining the execution sequence) in SFQ. Consider two threads A and B with weights 1 and 2, respectively, which become runnable at time $t = 0$. Let the scheduling quantum for each thread be 10ms and let $l_f = 10$. Let each thread consume the full length of the quantum each time it is scheduled. Initially, the virtual time $v(t) = 0$. Similarly, the start tags of threads A and B, denoted by S_A and S_B, respectively, are zero (i.e., $S_A = S_B = 0$). Since ties are broken arbitrarily, let us assume, without loss of generality, that thread A is scheduled first for one quantum. Since $v(t)$ is defined to be equal to the start tag of the packet in service, for $0 < t \leq 10 : v(t) = S_A = 0$. At the end of that quantum, the finish tag of A is computed as $F_A = 0 + \frac{10}{1} = 10$. Moreover, assuming that the thread remains runnable at the end of the quantum, it is stamped with $S_A = \max\{v(10), F_A\} = 10$. At this time, since $S_B < S_A$, the first quantum of thread B is scheduled. Note that since $S_B = 0$, the value of $v(t), 10 < t \leq 20$ continues to be equal to 0. At the end of this quantum, the finish tag for B is set to $F_B = 0 + \frac{10}{2} = 5$. Moreover, assuming that B remains runnable at the end of the quantum, we get $S_B = \max\{v(20), F_B\} = 5$. Carrying through this process illustrates that, before thread B blocks at time $t = 60$, threads A and B are scheduled for 20ms and 40ms, respectively, which is in proportion to their weights. When thread B is blocked, the entire CPU bandwidth is available to thread A, and the value of $v(t)$ changes at the beginning of each quantum of thread A. Now, when thread A blocks at time $t = 90$, the system contains no runnable threads. During this idle period, $v(t)$ is set to $\max\{F_A, F_B\} = \max\{50, 20\} = 50$. When thread A becomes runnable at time $t = 110$, $v(t) = 50$. Hence, thread A is stamped with $S_A = \max\{50, F_A\} = 50$, and is immediately schduled for execution. On the other hand, when thread B becomes runnable at time $t = 115$, $v(t) = S_A = 50$. Hence, it is stamped with $S_B = \max\{50, F_B\} = \max\{50, 20\} = 50$. From this point, the ratio of CPU allocation goes back to 1:2. Finally, when thread A exits the system, the entire CPU bandwidth becomes available to thread B, until it completes execution. Figure 3 illustrates this complete execution sequence.

3.1 Properties of SFQ

In what follows, we describe the properties of the SFQ scheduling algorithm, and demonstrate that it meets the requirements of a hierarchical scheduler outlined in the previous section.

1. *SFQ achieves fair allocation of CPU regardless of variation in available processing bandwidth* and hence meets the key requirement of a scheduling algorithm for hierarchical partitioning. Specifically, in [6], we have shown that regardless of fluctuations in available processor bandwidth, SFQ guarantees that in any interval $[t_1, t_2]$ in which two threads f

and m are eligible for being scheduled, the following inequality holds:

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m} \quad (3)$$

where l_f^{max} and l_m^{max}, respectively, are the maximum length of quantum for which threads f and m are scheduled[5]. It has been shown in [4] that if a scheduling algorithm schedules threads in terms of quantums and guarantees that $\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq H(f,m)$ for all intervals, then $H(f,m) \geq \frac{1}{2}(\frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m})$. Hence, SFQ is a near-optimal fair scheduling algorithm. In fact, no known algorithm achieves better fairness than SFQ.

2. *SFQ does not require the length of the quantum to be known a priori*: Since SFQ schedules threads in the increasing order of start tags, it does not need the length of the quantum of a thread to be known at the time of scheduling. The length of quantum q_f^j (namely, l_f^j) is required only when it finishes execution, at which time this information is always available. This feature is highly desirable in multimedia computing environments, where the computation requirements are not known precisely and threads may block for I/O even before they are preempted.

3. *SFQ provides bounds on maximum delay incurred and minimum throughput achieved by the threads in a realistic environment*: In most operating systems processing of hardware interrupts occurs at the highest priority. Consequently, the effective bandwidth of CPU fluctuates over time. SFQ provides bounds on delay and throughput even in such an environment. To derive these bounds, however, the variation in the CPU bandwidth has to be quantified. If the maximum rate of occurrence of interrupts and the CPU bandwidth used by the interrupts is known, the effective CPU bandwidth can be modeled as a Fluctuation Constrained (FC) server [11]. A FC server has two parameters; average rate C (instructions/s) and burstiness $\delta(C)$ (instructions). Intuitively, in any interval during a busy period, an FC server does at most $\delta(C)$ less work than an equivalent constant rate server. Formally,

Definition 1 *A server is a Fluctuation Constrained (FC) server with parameters $(C, \delta(C))$, if for all intervals $[t_1, t_2]$ in a busy period of the server, the work done by the server, denoted by $W(t_1, t_2)$, satisfies:*

$$W(t_1,t_2) \geq C*(t_2-t_1) - \delta(C) \quad (4)$$

If only bounds on the computation time required by the interrupts is known, then the FC server model is sufficient. However, if distributions of the computation time requirements for processing interrupts are known, then CPU is better modeled as an Exponentially Bounded Fluctuation (EBF) server [11]. An EBF server is a stochastic relaxation of FC server. Intuitively, the probability of work done by an EBF server deviating from the average rate by more than γ, decreases exponentially with γ. Formally,

Definition 2 *A server is an Exponentially Bounded Fluctuation (EBF) server with parameters $(C, B, \alpha, \delta(C))$, if for all intervals $[t_1, t_2]$ in a busy period of the server, the work done by the server, denoted by $W(t_1, t_2)$, satisfies:*

$$P\left(W(t_1,t_2) < C*(t_2-t_1) - \delta(C) - \gamma\right) \leq Be^{-\alpha\gamma} \quad (5)$$

If CPU can be modeled as an FC or EBF server, then SFQ provides throughput and delay guarantees to each of the threads. To determine these guarantees, let the weights of the threads be interpreted as the rate assigned to the threads. For example, a thread that needs 30% of a 100MIPS CPU would have a rate of 30 MIPS. Let Q be the set of threads served by CPU and let $\sum_{n\in Q} r_n \leq C$ where C is the capacity of the CPU. Then, SFQ provides the following throughput and delay guarantees [6]:

Throughput Guarantee: If the CPU is an FC server with parameters $(C, \delta(C))$, then the throughput received by a thread f with weight r_f is also FC with parameters:

$$\left(r_f, r_f\frac{\sum_{n\in Q} l_n^{max}}{C} + r_f\frac{\delta(C)}{C} + l_f^{max}\right) \quad (6)$$

If, on the other hand, the CPU is an EBF server with parameters $(C, B, \alpha, \delta(C))$, then the throughput received by a thread f with weight r_f is also EBF with parameters:

$$\left(r_f, B, \frac{r_f}{C}\alpha, r_f\frac{\sum_{n\in Q} l_n^{max}}{C} + r_f\frac{\delta(C)}{C} + l_f^{max}\right) \quad (7)$$

Hence, if SFQ is used for hierarchical partitioning and if the CPU is an FC(EBF) server, then each of

[5] The maximum quantum length may be known a-priori or may be enforced by a scheduler by preempting threads.

the sub-classes of the root class are FC(EBF) servers. Using this argument recursively, we conclude that if the CPU is an FC(EBF) server, then each of the sub-classes are also FC(EBF) servers, the parameters of which can be derived using (6) and (7).

Delay Guarantee: If the CPU is a FC server, then SFQ guarantees that the time at which quantum q_f^j of thread f will complete execution, denoted by $L_{SFQ}(q_f^j)$, is given as:

$$L_{SFQ}(q_f^j) \leq EAT(q_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} \quad (8)$$

where $EAT(q_f^j)$ is the expected arrival time of quantum q_f^j. Intuitively, $EAT(q_f^j)$ is the time at which quantum q_f^j would start if only thread f was in the system and the CPU capacity was r_f. Formally,

$$EAT(q_f^j) = \max\{A(q_f^j), EAT(q_f^{j-1}) + \frac{l_f^{j-1}}{r_f}\} \quad (9)$$

where $EAT(q_f^0, r_f^0) = -\infty$.

If the CPU is an EBF server, then SFQ guarantees that $L_{SFQ}(q_f^j)$ is given as follows:

$$P\left(L_{SFQ}(q_f^j) \leq EAT(q_f^j, r_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} + \frac{\gamma}{C}\right) \geq 1 - Be^{-\alpha\gamma} \quad 0 \leq \gamma \quad (10)$$

The following example illustrates the delay guarantee of SFQ.

Example 2 *Consider a constant rate 100MIPS CPU that serves threads* $1, 2,$ *and* 3. *Let thread* 1 *reserve 30MIPS and execute 300K instructions every quantum. Also, let the other two threads execute at most 200K instructions every quantum. Then, since* $\delta(C) = 0$ *for a constant rate CPU, for thread* 1, $\sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} = 7ms$. *Since executing 300K instructions on a 30MIPS CPU takes* $10ms$, *the expected arrival time of* j^{th} *quantum of thread* 1, *assuming it remains runnable at the end of each of its allocated quantum, is* $(j-1)*10ms$. *Hence, SFQ guarantees that* j^{th} *quantum of thread* 1 *will finish execution by* $(j-1)*10+7ms$.

Thus, SFQ not only guarantees fair allocation of CPU to sub-classes, but also provides quantitative bounds on performance.

4. *SFQ is computationally efficient*: Whereas the computation of a start tag only requires one addition and one division, sorting can be efficiently done using a priority queue. The computational complexity of a priority queue is known to $O(logQ)$, where Q is the number of entities to be scheduled. Since the number of children of a node in a hierarchy are expected to be small (of the order of 2-10), this cost is insignificant when SFQ is used for hierarchical partitioning. Furthermore, no other known algorithm that simultaneously achieves predictable allocation and protection has a lower complexity. Although static priority algorithms have lower complexity, they provide no protection, and hence, have been found to be unsatisfactory for multimedia operating systems [15].

Recall from Section 1 that a scheduling algorithm suitable for video applications should: (1) provide QoS guarantees even in presence of overload, and (2) not require computation requirements to be known precisely. Since SFQ guarantees fair allocation of resources even in presence of overload and does not need computation requirements to be known precisely, it meets these requirements. Hence, it suitable for video applications as well.

4 Implementation

We have implemented our hierarchical CPU allocation framework in the Solaris 2.4 kernel, which is a derivative of SVR4 UNIX. Our framework utilizes SFQ to schedule all the intermediate nodes for achieving hierarchical partitioning. The requirements of hierarchical partitioning are specified through a tree referred to as a *scheduling structure*. Each node in the scheduling structure has a weight, a start-tag, and a finish-tag that are maintained as per the SFQ algorithm. A non-leaf node maintains a list of child nodes, a list of runnable child nodes sorted by their start-tags, and a virtual time of the node which, as per SFQ, is the minimum of the start-tags of the runnable child nodes. A leaf node has a pointer to a function that is invoked, when it is scheduled by its parent node, to select one of its threads for execution. Each node also has a unique identity and a name similar to a UNIX filename. For example, in the scheduling structure shown in Figure 2, the name of node user1 is "/best-effort/user1". The scheduling structure is created using the following system calls:

- `int hsfq_mknod(char *name, int parent, int weight, int flag, scheduler_id sid)`: This system call creates a node with the given name as a child of node `parent` in the scheduling structure and returns the identifier of the new node. The `flag` parameter identifies the node to be created as a leaf or a non-leaf node. If the node is a leaf node, a pointer to the scheduling function of the class, identified by sid, is installed in the node.

- int hsfq_parse(char* name, int hint): This system call takes a name and resolves it to a node identifier in the scheduling structure. The name can be absolute or relative. If it is relative, it is considered to be relative to the node with identifier hint.

- int hsfq_rmnod(int id, int mode): This system call is used to remove a node from the scheduling structure. A node can be removed only if it does not have any child nodes.

- hsfq_move(int from, int to, ...): This system call is used to move a thread from one leaf node to another.

- hsfq_admin(int node, int cmd, void *args) : This system call is used for administration operations, other than those mentioned above, on the scheduling structure. Examples of administration operations include changing the weight of a node, determining the weight of a node, etc.

Given a scheduling structure, the actual scheduling of threads occurs recursively. To select a thread for execution, a function hsfq_schedule() is invoked. This function traverses the scheduling structure by always selecting the child node with the smallest start tag until a leaf node is selected. When a leaf node is selected, a function that is dependent on the leaf node scheduler, determined through the function pointer that is stored in the leaf node by hsfq_mknod(), is invoked to determine the thread to be scheduled. When a thread blocks or is preempted, the finish and the start tags of all the ancestors of the node to which the thread belongs have to be updated. This is done by invoking a function hsfq_update() with the duration for which the thread executed and the node identifier of the leaf node as parameters.

A node in the scheduling structure is scheduled if and only if at least one of the leaf nodes in the sub-tree rooted at that node has a runnable thread. The eligibility of a node is determined as follows. When the first thread in a leaf node becomes eligible for scheduling, function hsfq_setrun() is invoked with the leaf node's identifier. This function marks the leaf node as runnable and all the other ancestor nodes that may become eligible as a consequence. Note that this function has to traverse the path from the leaf up the tree only until a node that is already runnable is found. On the other hand, when the last thread in a leaf node makes a transition to sleep mode, function hsfq_sleep() is called with the leaf node's identifier. This function marks the leaf node as ineligible and all the other ancestor nodes that may become ineligible as a consequence. This function has to traverse the path from the leaf only until a node that has more than one runnable child nodes is found.

In our implementation, any scheduling algorithm can be used at the leaf node as long as it: (1) provides an interface function that can be invoked by hsfq_schedule() to select the next thread for execution, and (2) invokes hsfq_setrun(), hsfq_sleep(), and hsfq_update() as per the rules defined above. We have implemented SFQ as well as modified the existing SVR4 priority based scheduler to operate as a scheduler for a leaf node. The SVR4 leaf scheduler in our implementation, as in the standard release, uses a scheduling algorithm that is dependent on the scheduling class of a thread (e.g., time-sharing, interactive, system, etc.). Hence, in our implementation, a scheduler for a leaf node itself can use multiple scheduling policies.

Observe that the threads in a system may synchronize or communicate with each other, which can result in priority inversion (i.e., a scenario in which a lower priority thread may block the progress of a higher priority thread). The threads that synchronize/communicate may either belong to the same leaf class or different leaf classes. If the threads belong to different leaf classes, the notion of priority inversion is not defined. Furthermore, synchronization between threads belonging to different classes is not desirable, since that may lead to violation of QoS requirements of applications. For example, if a thread in the real-time leaf class synchronizes with a thread in the best-effort class, then, since the best-effort class does not perform any admission control, the QoS requirement of the thread may be violated. Techniques for avoiding priority inversion among threads belonging to the same leaf class, on the other hand, depend on the leaf class scheduler. For example, if the leaf scheduler uses static priority Rate Monotonic algorithm, then standard priority inheritance techniques can be employed [13, 14]. Similarly, when the leaf scheduler is SFQ, priority inversion can be avoided by transferring the weight of the blocked thread to the thread that is blocking it. Such a transfer will ensure that the blocking thread will have a weight (and hence, the CPU allocation) that is at least as large as the weight of the blocked thread.

We envision that our scheduling infrastructure would be used by a QoS manager [10] in a multimedia system (see Figure 4). Applications will specify their QoS requirements to the QoS manager which would: (1) determine the resources needed to meet the QoS requirements of the applications; (2) decide the scheduling class the application should belong to, and create the class if it does not exist; (3) employ class dependent admission control procedures to determine if the resource requirements can be satisfied (some classes may have no admission con-

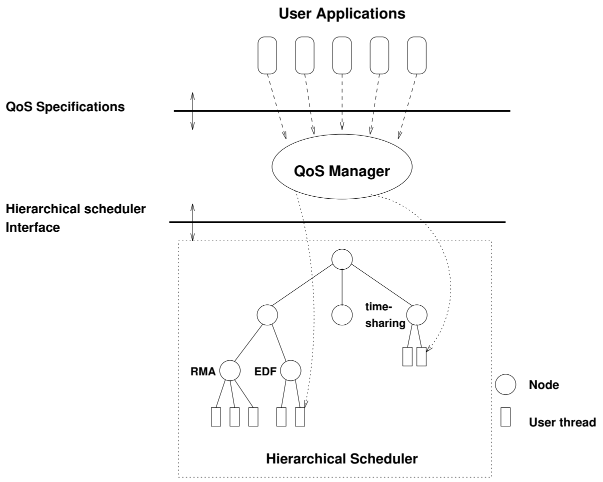

Figure 4 : Quality of Service Manager

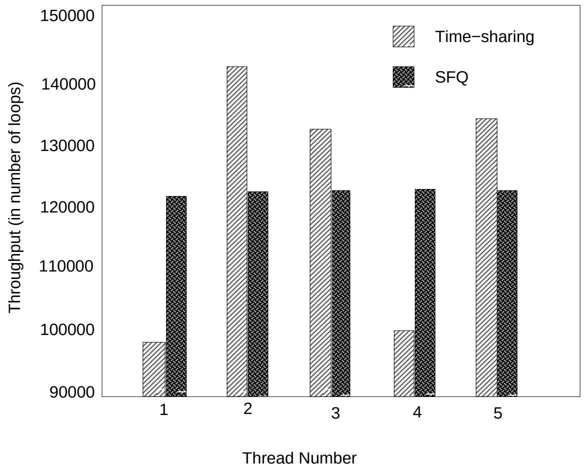

Figure 5 : Comparison of throughput of threads under SFQ and time-sharing schedulers

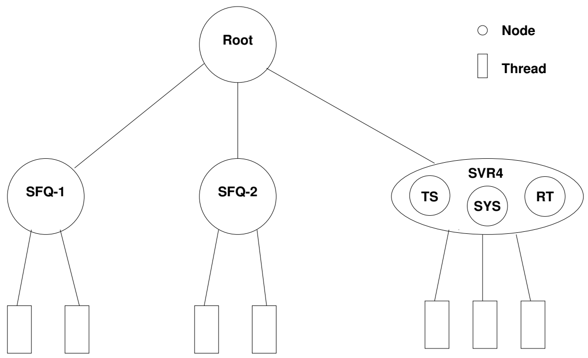

Figure 6 : Scheduling structure used for the experiments

trol); and (4) allocate the resources to the application and move it to appropriate class. The QoS manager may also move applications between classes or change the resource allocation in response to change in QoS requirements. It would also dynamically change the relative allocations of different classes so as to effectively meet the requirements of the applications that may coexist at any time. The development of such policies is the subject of future research and beyond the scope of this paper.

5 Experimental Evaluation

We have evaluated the performance of our implementation using a Sun SPARCstation 10 with 32MB RAM. All our experiments were conducted in multiuser mode with all the normal system processes. Most of our experiments were carried out using the Dhrystone V2.1 benchmark, which is a CPU intensive application that executes a number of operations in a loop. The number of loops completed in a fixed duration was used as the performance metric. We evaluated several aspects of the hierarchical scheduler, the results of which are reported in the following sections.

5.1 Limitation of Conventional Schedulers

We had argued that conventional time-sharing schedulers are inadequate for achieving predictable resource allocation in multimedia operating systems. To experimentally validate this claim, we compared the throughput of 5 threads running Dhrystone benchmark under time-sharing and SFQ schedulers. Whereas in the case of SFQ all the threads had equal weight, in the case of time-sharing scheduler all the threads were assigned the same initial user priority. Figure 5 demonstrates that, in spite of having the same user priority, the throughput received by the threads in the time-sharing scheduler varies significantly, thereby demonstrating it's inadequacy in achieving predictable allocation. In contrast, all the threads in SFQ received the same throughput in conformance with the theoretical predictions. In [15], it has been demonstrated that when a multimedia application is run as a real-time thread in the SVR4 scheduler, the whole system may become unusable. This limitation of the SVR4 scheduler coupled with the unpredictability of time-sharing algorithm clearly demonstrates the need for a predictable scheduling algorithm for multimedia operating system.

5.2 Scheduling Overhead

A key concern in using dynamic priority-based algorithm such as SFQ is that the scheduling overhead may be high. To evaluate the overhead, we determined the ratio of the number of loops completed by a thread in our hierarchical scheduler and the unmodified kernel. In the hierarchical scheduler, we used the scheduling structure shown in Figure 6 with the threads belonging to node SFQ-1. To determine the effect of the number of threads on the scheduling overhead, the number of threads executing the Dhrystone benchmark was varied from 1 to 20. Figure 7(a) plots the variation in the ratio of the aggregate throughput of threads in our hierarchical scheduler to that in the unmodified kernel against the increase in the number of threads. The ratio was determined by averaging over 20 runs and using a time quantum of 20ms. As Figure 7(a) demonstrates, the throughput achieved by our scheduler is within 1% of the throughput of the unmodified kernel.

To evaluate the impact of the depth of the scheduling structure, the number of nodes between the root class and the SFQ-1 class was varied from 0 to 30. As Figure 7(b) demonstrates, in spite of the significant variation in the depth, the throughput remains within 0.2%. These experiments demonstrates that it is feasible to employ SFQ for hierarchical CPU scheduling.

5.3 Hierarchical CPU allocation

We evaluated the effectiveness of SFQ in achieving hierarchical CPU allocation using the scheduling structure shown in Figure 6. Nodes SFQ-1, SFQ-2 and SVR4 were assigned weights of 2, 6, and 1, respectively. Two threads executing the Dhrystone benchmark were added to leaf nodes SFQ-1 and SFQ-2 (SVR4 node contained all the other threads in the system). Figure 8(a) demonstrates that the aggregate throughput of nodes SFQ-1 and SFQ-2 (measured in terms of number of completed loops of the Dhrystone benchmark) are in the ratio 1:3 (i.e., in accordance to their weights). Observe that due to the variation in the CPU usage of the threads belonging to node SVR4, the aggregate throughput of nodes SFQ-1 and SFQ-2 fluctuates over time. In spite of this variation, nodes 1 and 2 receive throughput in the ratio 1:3, thereby demonstrating the SFQ achieves fair allocation even when the available CPU bandwidth fluctuates over time.

A key advantage of our hierarchical scheduler is that even though different leaf schedulers may be used, each node receives its fair allocation and is isolated from the other nodes. To demonstrate this, we used the scheduling structure shown in Figure 6 with 2 threads in SFQ-1 and 1 thread in SVR4. SFQ-1 as well SVR4 nodes were assigned the same weight. Figure 8(b) demonstrates that the threads in SFQ-1 node as well as SVR4 node make progress and are isolated from each other. Furthermore, both SFQ-1 and SVR4 nodes receive the same throughput. This is in contrast to the standard SVR4 scheduler where a higher priority class, such as the real-time class, can monopolize the CPU.

To demonstrate the feasibility of supporting hard real-time applications in our hierarchical scheduling framework, we used the scheduling structure shown in Figure 6, and executed two threads (namely, thread1 and thread2) in the RT class of the SVR4 node, and an MPEG decoder in SFQ-1 node. The SVR4 and the SFQ-1 nodes were given equal weights. Whereas thread1 executed for 10 ms every 60 ms, thread2 required 150 ms of computation time every 960 ms. Rate monotonic algorithm was used to schedule these two threads. For each thread, a clock interrupt was used to announce the deadline for the current round and the start of a new round of computation. The threads were scheduled for 25ms quantums. We measured the performance of the system using two parameters: (1) *scheduling latency*, which refers to the duration for which a thread has to wait prior to getting access to the CPU after its clock interrupt; and (2) *slack time*, which refers to the difference in time between the deadline and the time at which the current round of computation completes. Figure 9 depicts the variation in scheduling latency and slack time for each round for thread1. Whereas Figure 9(a) illustrates that thread1 gained access to the CPU within a bounded period of time (equal to the length of the scheduling quantum) after its clock interrupt, Figure 9(b) demonstrates that none of the deadlines for thread1 were violated (i.e., the slack time is always positive).

5.4 SFQ as a Leaf Scheduler

To evaluate the use of SFQ as a leaf scheduler, two threads with weights 5 and 10, each running the Berkeley MPEG video player, were assigned to node SFQ-1. Figure 10 plots the number of frames decoded by each thread as a function of time. It demonstrates that the thread with weight 10 decodes twice as many frames as compared to the other thread in any time interval.

5.5 Dynamic Bandwidth Allocation

A QoS manager may dynamically change the bandwidth allocation of classes to meet the application requirements. Hence, SFQ should be able to achieve fair allocation even when bandwidth allocation is dynamically varied. To evaluate this aspect of SFQ, two threads, each executing the Dhrystone benchmark, were run in the SFQ-1 node. The behavior of the threads was varied over time as follows:

- At time 0, both threads were assigned a weight of 4. Hence, the throughput ratio between threads was 4:4.
- At time 4, the weight of thread 2 was changed to 2. Hence, the throughput ratio became 4:2.

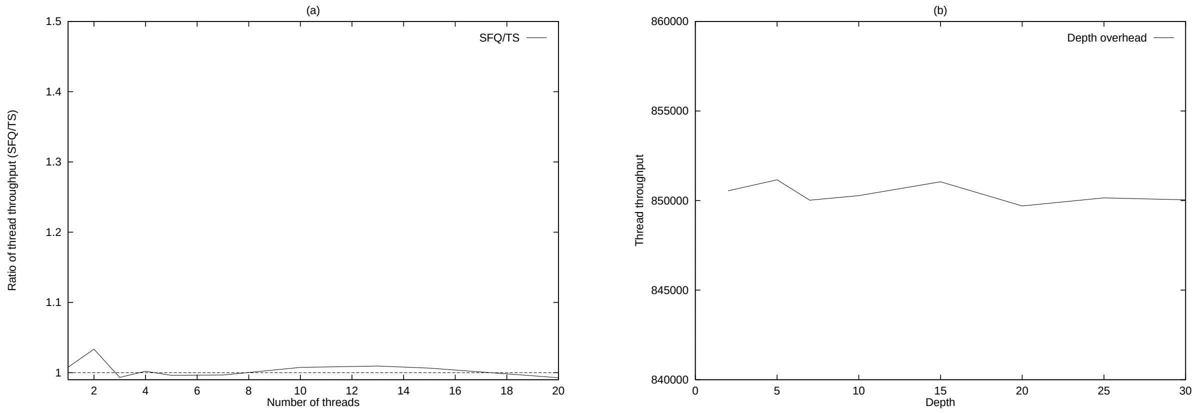

Figure 7 : (a) Ratio of number of loops executed in hierarchical and unmodified scheduler; (b) Variation in throughput with increase in depth of hierarchy

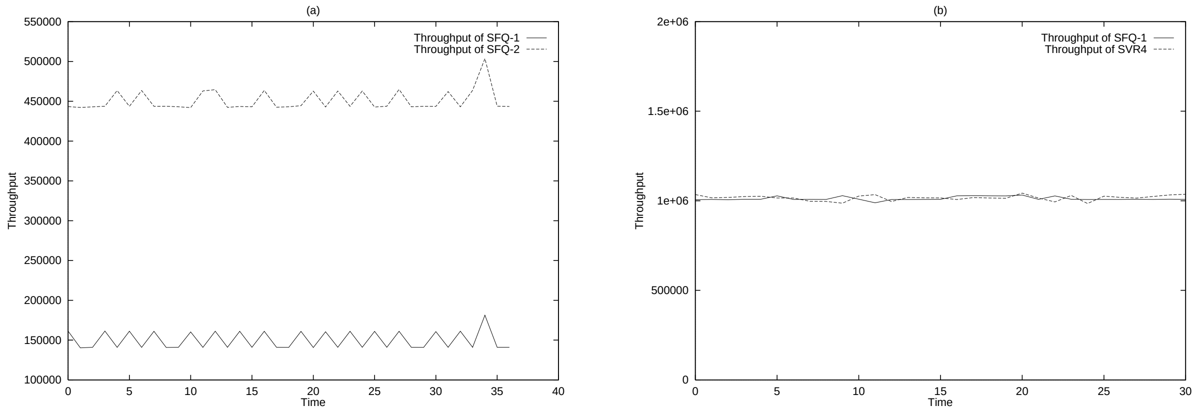

Figure 8 : (a) Aggregate throughput of nodes SFQ-1 and SFQ-2; (b) Throughput of nodes SFQ-1 and SVR4

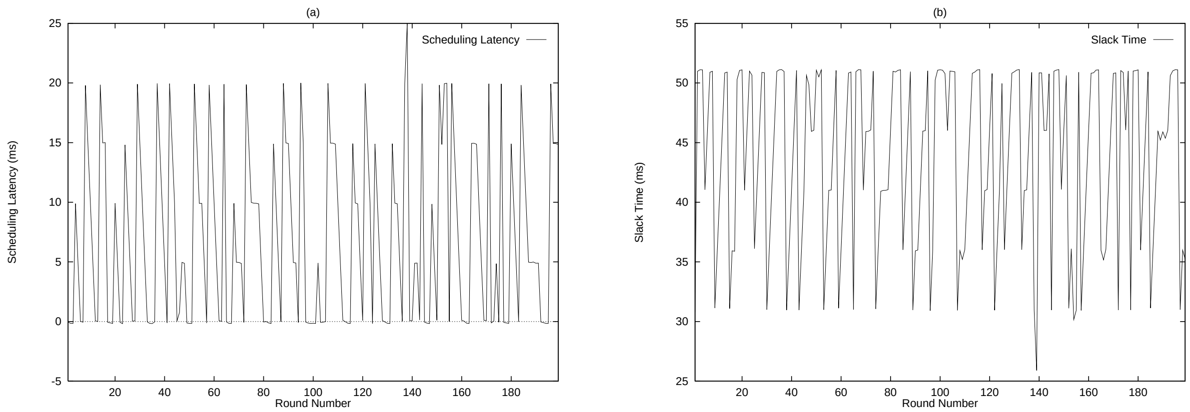

Figure 9 : Variation in: (a) scheduling latency and (b) slack time

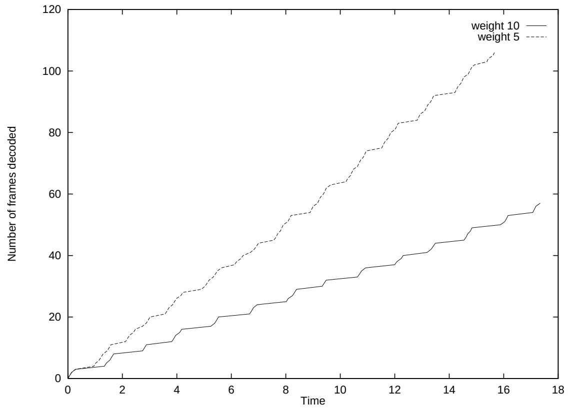

Figure 10 : Number of frames decoded as a function of time

- At time 6, thread 1 was put to sleep. Hence, the throughput ratio became 0:2.
- At time 9, thread 1 resumed execution. Hence, the throughput ratio became 4:2.
- At time 12, the weight of thread 1 was changed to 8. Hence, the throughput ratio became 8:2.
- At time 16, the weight of thread 2 was changed to 4. Hence, the throughput ratio became 8:4.
- At time 22, the weight of thread 1 was changed to 4. Hence, the throughput ratio became 4:4.

Figures 11(a) and 11(b), respectively, illustrate that the throughput of the threads (measured in terms of number of completed loops) and their ratio varies as per the changes in the weights of the threads. This demonstrates that SFQ can achieve fairness even in the presence of dynamic variation in weight assignments.

6 Related Work

We are not aware of any CPU scheduling algorithm that achieves hierarchical partitioning while allowing different schedulers to be used for different applications. However, since a fair scheduling algorithm is the basis for achieving hierarchical partitioning, we discuss other such algorithms proposed in the literature. Most of these algorithms have been proposed for fair allocation of network bandwidth; we have modified their presentation appropriately for CPU scheduling.

The earliest known fair scheduling algorithm is Weighted Fair Queuing (WFQ) [3]. WFQ was designed to emulate a hypothetical weighted round robin server in which the service received by each thread in a round is infinitesimal and proportional to the weight of the thread. Since threads can only be serviced in quantums at a time, WFQ emulates a hypothetical server by scheduling threads in the increasing order of the finishing times of the quantums of the threads in the hypothetical server. To compute this order, WFQ associates two tags, a *start tag* and a *finish tag*, with every quantum of a thread. Specifically, the start tag $S(q_f^j)$ and the finish tag $F(q_f^j)$ of quantum q_f^j are defined as:

$$S(q_f^j) = \max\{v(A(q_f^j)), F(q_f^{j-1})\} \quad j \geq 1 \tag{11}$$

$$F(q_f^{j-1}) = S(q_f^{j-1}) + \frac{l_f^{j-1}}{r_f} \quad j \geq 1 \tag{12}$$

where $F(q_f^0) = 0$ and $v(t)$ is defined as the round number that would be in progress at time t in the hypothetical server. Formally, $v(t)$ is defined as:

$$\frac{dv(t)}{dt} = \frac{C}{\sum_{j \in B(t)} r_j} \tag{13}$$

where C is the capacity of the CPU measured in instructions/second and $B(t)$ is the set of runnable threads at time t in the hypothetical server. WFQ then schedules quantums in the increasing order of their finish tags. WFQ has several drawbacks for scheduling a CPU:

- As demonstrated in [6], WFQ does not provide fairness when the processor bandwidth fluctuates over time. Since fairness in the presence of variation in available CPU bandwidth is crucial for supporting hierarchical partitioning, WFQ is unsuitable for a CPU scheduler in a general purpose operating system.
- WFQ requires the length of the quantums to be known a priori. Though the maximum length of the quantum may always be known (as the scheduler can enforce it by preempting a thread), for environments in which the computation requirements are not known precisely, the exact quantum length may not be known. If WFQ assumes the maximum quantum length for scheduling and if the thread uses less than the maximum, the thread will not receive its fair share. On the other hand, if WFQ is modified to reflect the actual length of the execution (by changing the finish tag of a quantum after the end of its execution), then WFQ would have been modified in a non-trivial manner. Though WFQ is known to have bounded fairness, it is not known if the modified algorithm retains its fairness properties.
- WFQ requires the computation of $v(t)$, which, in turn, requires simulation of the hypothetical server. This simulation is known to be computationally expensive [4]. In contrast, SFQ computes the start and the finish tags efficiently.

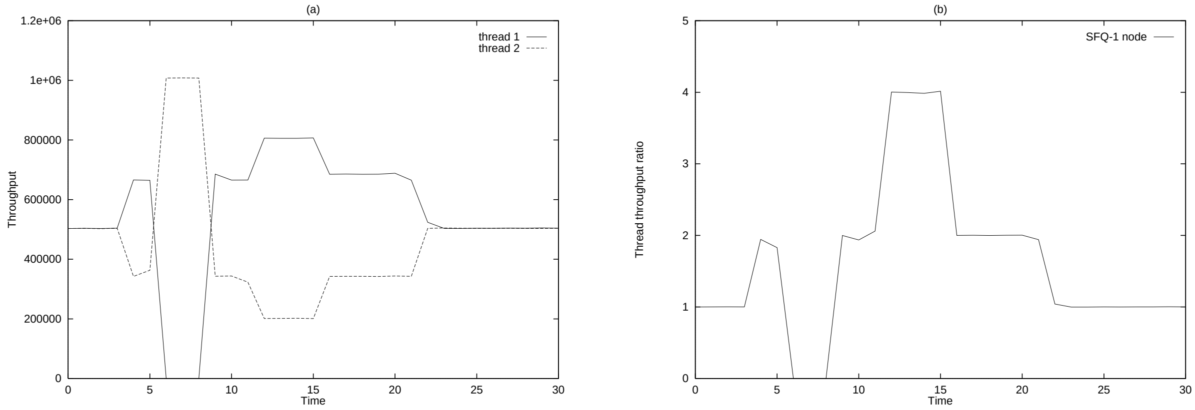

Figure 11 : (a) Throughput of threads 1 and 2; (b) Ratio of throughputs of threads 1 and 2

- The unfairness of WFQ, as derived in [16], is significantly higher than SFQ.

- WFQ provides high delay to low throughput applications. Specifically, it guarantees that quantum q_f^j will complete execution by:

$$EAT(q_f^j) + \frac{l_f^j}{r_f} + \frac{l^{max}}{C} \qquad (14)$$

  where l^{max} is the maximum quantum length ever scheduled at the CPU. Hence, using (8), we conclude that the difference in maximum delay incurred in SFQ and WFQ, denoted by $\Delta(q_f^j)$, is given as:

$$\Delta(q_f^j) = \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} - \frac{l_f^j}{r_f} - \frac{l^{max}}{C} \quad (15)$$

  Now, if all quantums are of the same lengths, then $\Delta(q_f^j) < 0$ (i.e., SFQ provides a better delay guarantee) if $r_f \leq \frac{1}{|Q|-1}$. Since this condition is expected to hold for low throughput applications, we conclude that SFQ provides lower delay to low throughput applications. Since interactive applications are low throughput in nature, this feature of SFQ is highly desirable for CPU scheduling.

Fair Queuing based on Start-time (FQS) was proposed in [7] to make WFQ suitable for CPU scheduling when quantum length may not be known a priori. It computes the start tag and the finish tag of a quantum exactly as in WFQ. However, instead of scheduling quantums in the increasing order of finish tags, it schedules them in the increasing order of start tags. Since quantum length is not required for computing the start tag, it becomes suitable for CPU scheduling. However, its main drawbacks are that: (1) just as WFQ, it is computationally expensive, and (2) it does not provide fairness when the available CPU bandwidth fluctuates over time, and consequently is unsuitable for hierarchical partitioning. Furthermore, it is not known to have any better properties than SFQ.

Self Clocked Fair Queuing (SCFQ), originally proposed in [2] and later analyzed in [4], was designed to reduce the computational complexity of fair scheduling algorithms like WFQ. It achieves efficiency over WFQ by approximating $v(t)$ with the finish tag of the quantum in service at time t. However, since SCFQ also schedules quantums in increasing order of finish tags, it is unsuitable for scheduling CPU in a multimedia operating system. Furthermore, although it has the same fairness and implementation complexity as SFQ, it provides significantly larger delay guarantee than SFQ. Specifically, it increases the maximum delay of quantum q_f^j by $\frac{l_f^j}{r_f}$ [6].

In the OS context, a randomized fair algorithm, termed lottery scheduling, was proposed in [19]. Due to its randomized nature, lottery scheduling achieved fairness only over large time-intervals. This limitation was later addressed by stride scheduling algorithm [18]. The stride scheduling algorithm is a variant of WFQ and, consequently, has all the drawbacks of WFQ. Furthermore, no theoretical properties of the stride scheduling algorithm are known. Recently, a proportionate share resource allocation algorithm, referred to as Earliest Eligible Virtual Deadline First (EEVDF), has been proposed [17].

Hierarchical partitioning of resource allocation was also proposed in [19] using the abstraction of tickets and currencies. In that framework, a thread is allocated tickets in some currency and the currency, in turn, is funded in terms of tickets of some other currency. The "funding" relationship is such that the value of a ticket in every currency can be translated to a value in the base currency. Every thread then is allocated resources in proportion to the value of its tickets in the base currency using lottery scheduling. This achieves hierarchical partitioning since if a thread becomes inactive, the value of the tickets of the threads that are funded by the same currency increases. This specification of hierarchical partitioning is similar to our scheduling structure. However, the key differences between our framework and the approach of [19] are as follows. First, our framework permits different scheduling algorithms to be employed for different classes of applications, whereas the framework of [19] does not. Second, hierarchical partitioning is achieved in [19] by re-computation of ticket values of every thread that are funded in the same currency or some ancestor of the currency every time a thread gets blocked or exits. This approach not only incurs additional overhead of computing ticket values, but also does not provide any guarantees. Hence, the requirements of hard and soft real-time applications can not be met in this framework. In contrast, our framework achieves hierarchical partitioning through a theoretically sound hierarchical scheduler.

Several other efforts have investigated scheduling techniques for multimedia systems [1, 5, 8, 13]. These scheduling algorithms are complementary to our hierarchical scheduler and can be employed as leaf class scheduler in our framework. Most of these algorithms require precise characterization of resource requirements of a task (such as computation time and period) as well as admission control to achieve predictable allocation of CPU. In contrast, SFQ requires neither of these; it just requires relative importance of tasks (expressed by weights) to be known. It requires admission control only if the applications desire a certain guaranteed minimum CPU bandwidth. Thus, SFQ allows a range of control over CPU allocation: whereas admission control can be used to guarantee a certain minimum CPU allocation to tasks and thus match the performance of the existing algorithms, admission control can be avoided when applications only require relative resource allocation. Such a flexibility is highly desirable in multimedia systems and the lack of it is one of the main disadvantage of existing algorithms. A detailed experimental investigation of the relative merits of these algorithms vis-a-vis SFQ as a leaf class scheduler is the subject of our current research.

7 Concluding Remarks

In this paper, we presented a flexible framework for hierarchical CPU allocation, using which an operating system can partition the CPU bandwidth among various application classes, and each application class, in turn, can partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. We presented the Start-time Fair Queuing (SFQ) algorithm, which enabled such hierarchical partitioning. We demonstrated that SFQ is not only suitable for hierarchical partitioning, but is also suitable for video applications. We have implemented the hierarchical scheduler in Solaris 2.4. We demonstrated that our framework: (1) enables co-existence of heterogeneous schedulers, (2) protects application classes from each other, and (3) does not impose higher overhead than conventional time-sharing schedulers. Thus, our hierarchical scheduling framework is suitable for multimedia operating systems.

References

[1] D. P. Anderson. Metascheduling for Continuous Media. *ACM Transactions on Computer Systems*, 11(3):266–252, August 1993.

[2] J. Davin and A. Heybey. A Simulation Study of Fair Queueing and Policy Enforcement. *Computer Communication Review*, 20(5):23–29, October 1990.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[4] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94*, 1994.

[5] R. Govindan and D. P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 68–80, October 1991.

[6] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

[7] A. Greenberg and N. Madras. How Fair is Fair Queuing. *The Journal of ACM*, 39(3):568–598, July 1992.

[8] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications*, 15:388–395, July/August 1992.

[9] K. Jeffay and D.L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium Raleigh-Durham, NC*, pages 212–221, December 1993.

[10] M.B. Jones, P. Leach, R. Draves, and III J. Barrera. Support for User-Centric Modular Real-Time Resource Management in Rialto Oper-

ating System. In *Proceedings of NOSSDAV'95, Durham, New Hampshire*, April 1995.

[11] K. Lee. Performance Bounds in Communication Networks With Variable-Rate Links. In *Proceedings of ACM SIGCOMM'95*, pages 126–136, 1995.

[12] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprocessing in a Hard-Real Time Environment. *JACM*, 20:46–61, January 1973.

[13] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, May 1994.

[14] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the 14th IEEE Real-Time Systems Symp.*, pages 120–130, December 1993.

[15] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 41–53, November 1993.

[16] D. Stiliadis and A. Varma. Design and Analysis of Frame-based Fair Queueing: A New Traffic Scheduling Algorithm for Packet Switched Networks. In *Proceedings of SIGMETRICS'96*, May 1996.

[17] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium (to appear)*, December 1996.

[18] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[19] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposim on Operating System Design and Implementation*, November 1994.