A Hierarchical Scheduler for Multimedia Operating Systems

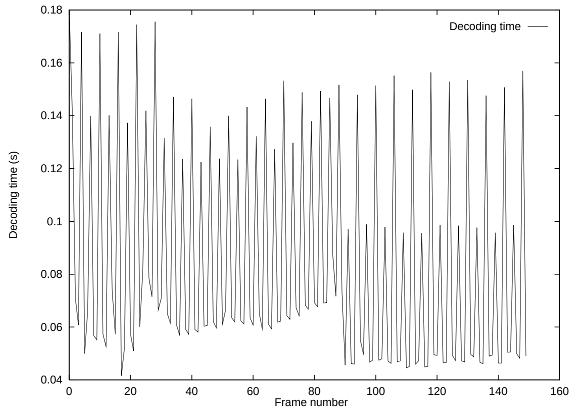
Pawan Goyal, Xingang Guo, and Harrick M. Vin

Distributed Multimedia Computing Laboratory
Department of Computer Sciences
University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188
URL: http://www.cs.utexas.edu/users/dmcl

October 30, 1996

Application Requirements

- Hard real-time applications: deterministic guarantees
- Soft real-time applications: statistical guarantees
  - Example: VBR video applications



- Best-effort applications: no absolute performance guarantees, but low average response times

Supporting Multiple Application Classes

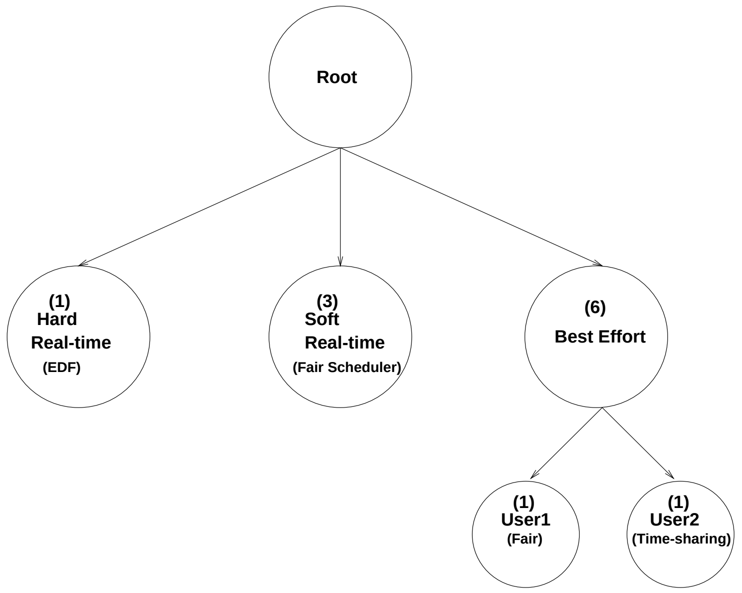
- Different application classes $\Rightarrow$ different scheduling algorithms
- Need an OS framework that:
  - Facilitates co-existence
  - Provides protection between various application classes (e.g., overbooking CPU for soft real-time should not affect hard real-time applications)
- Solution: Hierarchical partitioning of CPU bandwidth

Hierarchical CPU Scheduling

- Hierarchical partitioning specified as a *tree*

- Leaf nodes:
  - Aggregation of threads
  - Scheduled by application-specific scheduler

- Intermediate nodes:
  - Aggregation of application classes
  - Scheduled by an algorithm that achieves hierarchical partitioning



Requirements of a Hierarchical CPU Scheduler

- Should achieve proportionate allocation of CPU bandwidth allocated to a class among its sub-classes, even when the bandwidth available to a class fluctuates over time
- Should not require computational requirements of tasks to be known a priori
- Should provide throughput and delay guarantees
- Should be computationally efficient

Proportionate Allocation

- Proportionate allocation: tasks receive CPU bandwidth in proportion to weights

- Ideal definition:

$$\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$$

$W_f(t_1,t_2)$	:	aggregate work done by thread f in interval in $[t_1,t_2]$
r_f	:	weight of thread f

- Quantum-based scheduling:

$$\left| \frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} \right| \leq H(f,m)$$

- $H(f,m)$: fairness measure

- Objective: achieve small fairness measure

Existing Techniques

- Simple approach: Weighted Round Robin scheduling
  - Achieves proportionate allocation only at large time scales
  - Does not support variable length quantums
- Large body of literature in networks and OS
- Weighted Fair Queuing (WFQ):
  - Unfair when processor bandwidth fluctuates over time
  - Requires length of quantums to be known a priori
  - Computationally expensive

Start-time Fair Queuing (SFQ)

- Start tag S_f and finish tag F_f:

$$S_f = \max\{v(A(q_f^j)), F_f\}$$

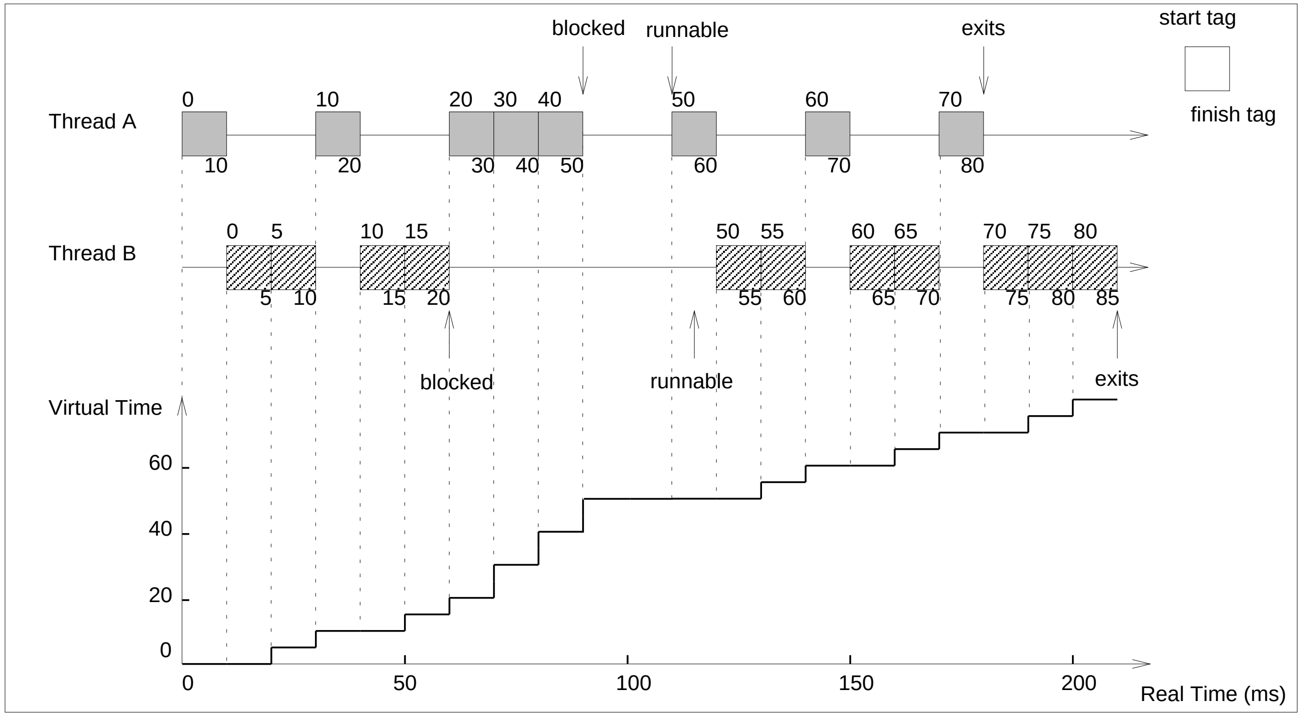
$$F_f = S_f + \frac{l_f^j}{r_f}$$

q_f^j	:	j^{th} quantum of thread f
l_f^j	:	length of q_f^j
$A(q_f^j)$	:	time at which the j^{th} quantum is requested
r_f	:	weight of thread f

- Virtual time $v(t)$: start tag of the thread in service at time t
- Threads are service in the increasing order of start tags

SFQ: An Example

- Computation of virtual time, start tag, and finish tag in SFQ



Properties of SFQ

- SFQ achieves fair allocation of CPU regardless of variation in available processing bandwidth

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

- SFQ does not require the length of the quantum to be known a priori
- SFQ provides bounds on maximum delay incurred and minimum throughput achieved by threads in realistic environments
- SFQ is computationally efficient

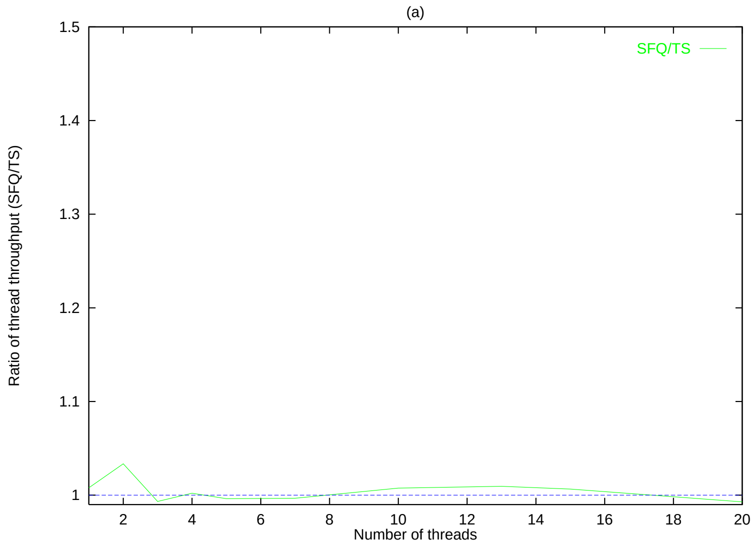
Implementation

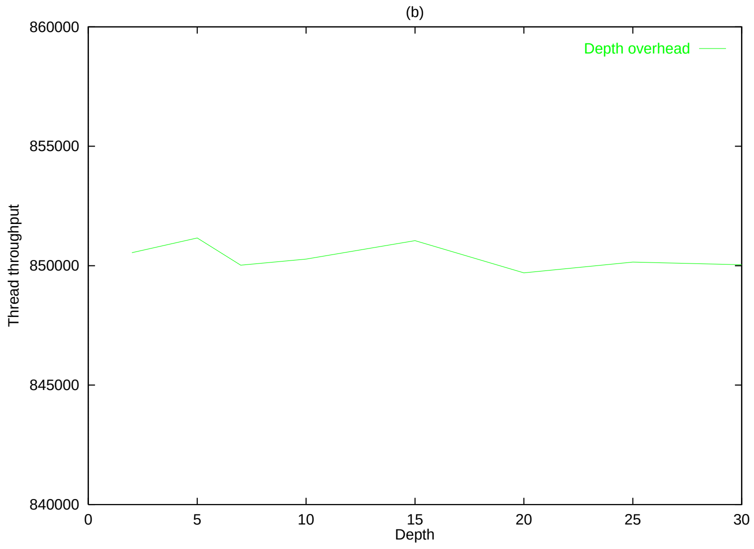
- Implemented hierarchical CPU allocation framework in Solaris 2.4 kernel
- System calls to create scheduling structure
  - int hsfq_mknod(char *name, int parent, int weight, int flag, scheduler_id sid)
  - int hsfq_parse(char* name, int hint)
  - int hsfq_rmnod(int id, int mode)
  - hsfq_move(int from, int to, ...)
  - hsfq_admin(int node, int cmd, void *args)

Implementation (Contd.)

- Scheduling done hierarchically
- hsfq_update(): to update start and finish tags of all the nodes in the hierarchy
- hsfq_setrun(): when the **first** thread in a leaf node becomes active
- hsfq_sleep(): when the **last** thread in a leaf node sleeps
- Priority inversion:
  - Synchronization should occur within the same leaf node
  - Inversion within leaf node avoided by using leaf scheduler dependent policies
    - Priority transfer in case of RMA
    - Weight transfer in case of SFQ

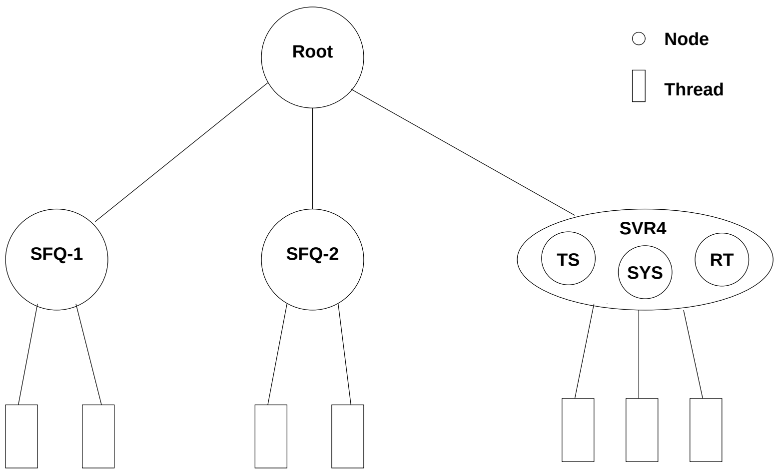
Scheduling Overhead



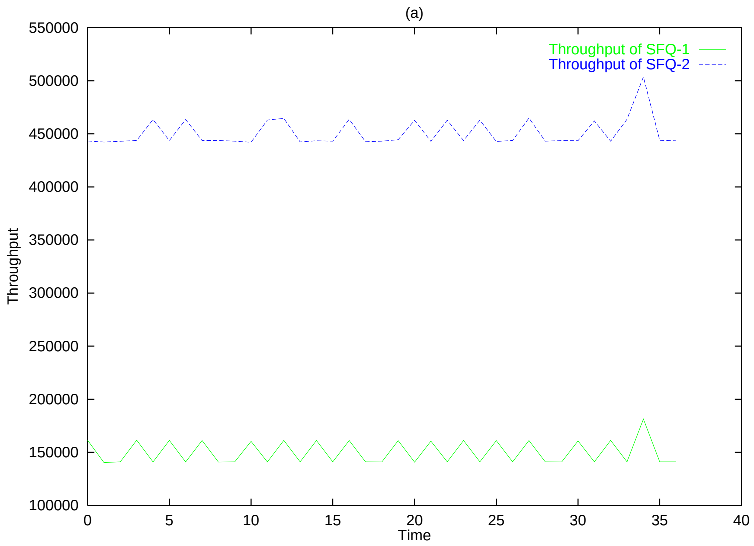


- Scheduling overhead is negligible

Hierarchical CPU allocation

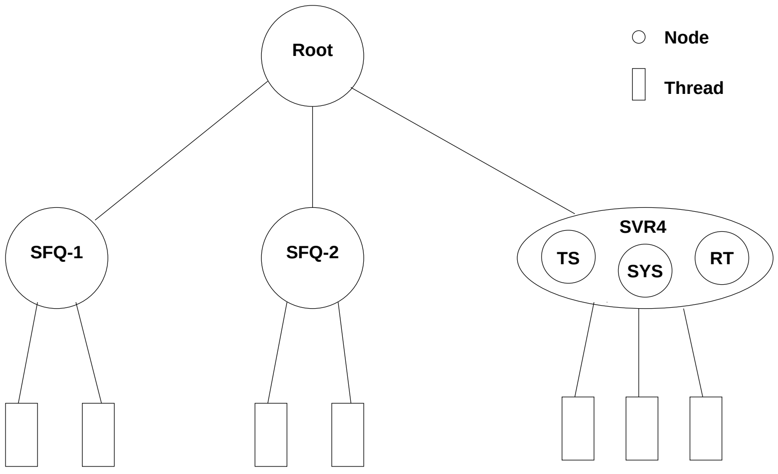


Scheduling structure with weights 2, 6, and 1

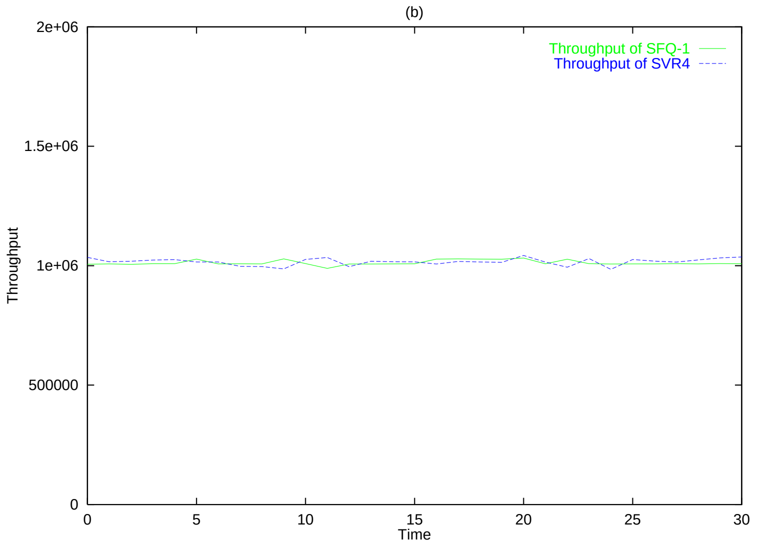


Throughput

Coexistence of Multiple Schedulers



Scheduling structure with equal weights to SFQ-1 and SVR4



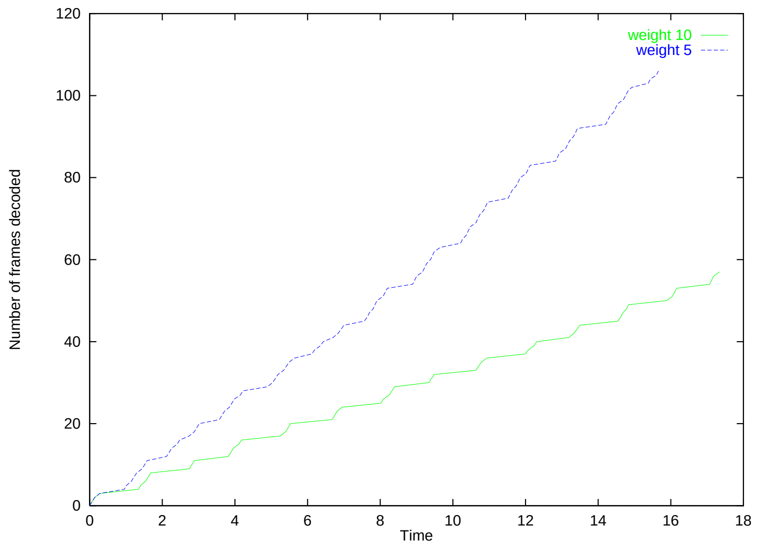
Throughput

Supporting Hard Real-time applications

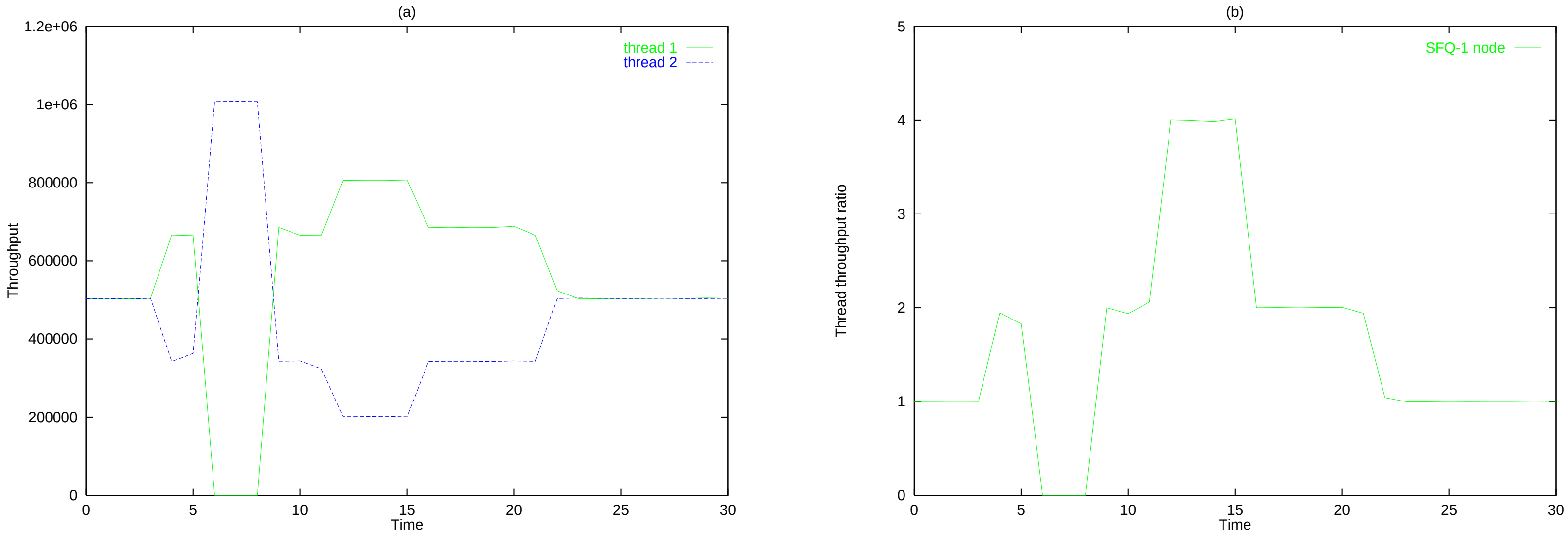
- Two threads in RT class of SVR4 node
  - Thread 1: 10 ms computation every 60 ms
  - Thread 2: 150 ms computation every 960 ms
- RMA used to schedule Thread 1 and Thread 2
- Threads scheduled for 25ms quantums
- Results:
  - Threads gained access to CPU within bounded period of time after their clock interrupt
  - None of the deadlines were violated

SFQ as a Leaf Scheduler

- MPEG decoders with weights 1 and 2



Dynamic Bandwidth Allocation



- SFQ achieves proportionate allocation even in the presence of dynamic variation in weight assignment

Concluding Remarks

- Start-time Fair Queuing (SFQ):
  - Hierarchical partitioning
  - Soft real-time video applications
- Hierarchical scheduler implemented in Solaris 2.4
- Flexible framework for hierarchical CPU allocation
  - Enables co-existence of heterogeneous schedulers
  - Protects application classes from each other
  - Does not impose any higher overhead than conventional time-sharing schedulers