Scheduling Multimedia Applications under Overload and Indeterministic Conditions *

Teik Guan Tan and Wynne Hsu
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore 119260
(65)-772-7944
{tanteikg, whsu}@iscs.nus.sg

Abstract

Scheduling multimedia applications is not straightforward. The deterministic techniques in hard real-time scheduling, though applicable, are not absolutely necessary and instead result in very conservative execution. Current real-time scheduling research does not properly address the problem of scheduling under overload and nondeterministic situations. In this paper, we introduce a more representative task model for multimedia applications. We describe the use of a frequency parameter, associated with the tasks, which can serve to regulate execution during degradation (when the system is overloaded). We also analyze the worst case CPU bounds for scheduling using frequency and show the possibility of utilization, $\sum \frac{c}{p} > 1$ under certain conditions. We conduct experiments to see the performance of the model and best-effort frequency-based scheduling under extreme and variable conditions, where modeling fails.

1 INTRODUCTION

Scheduling is the art of sequencing a set of tasks to execute in a correct order to achieve a goal. For hard real-time scheduling, the goal is normally such that *no task will miss its deadline*. However, if execution is not hard real-time, the open question is then what is the goal? Figure 1 which describes a simple scenario of two tasks, T_a and T_b, and the different possibilities of scheduling them. One glaring point made in Figure 1 is that there are still many questions left unanswered concerning how should scheduling be performed when not all tasks can be executed to meet their deadlines. Most real-time scheduling models [11, 13] concentrate on how to schedule tasks when there are sufficient resources. The case is different under overload conditions, when not all tasks can be scheduled on time and hence some form of execution tradeoff is necessary. For multimedia applications, there is no absolute need for all the tasks to be executed within the deadline all the time. This is because execution requirements for such applications are rarely life-threatening (vis-a-vis hard real-time systems) and a lower-than-optimal quality output is often acceptable due to the large CPU and resource demands required to service multimedia task execution. This uncertainty during overloads is not only difficult to model and manage, the criteria for deciding the degree of quality to be sacrificed during degradation is also vague. The problem of scheduling multimedia applications is further complicated by the non-static nature of multimedia data. The amount of time needed to compress/decompress a video image may vary widely depending on the content of the image.

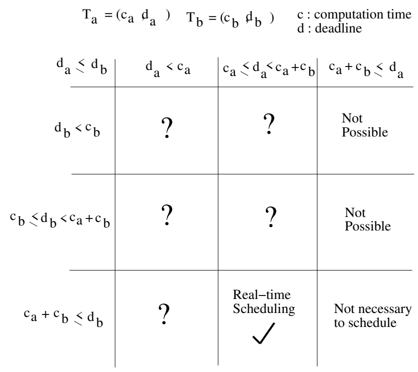

Figure 1: Scheduling decisions for two tasks

In this paper, we present a multimedia task model to structure the real-time needs of multimedia applica-

*This work was supported in part by the NSTB-MOE of Singapore under Research Project Grant PS964770

tions. We also discuss ways to manage the degradation through the use of frequencies. A point to note here is that many scheduling problems under overload conditions are NP-Complete [3] and hence, our model cannot rely solely on the scheduling algorithm to provide good results. Instead, the user participates in controlling the degradation of the application by structuring the application into suitable partitions and specifying the acceptable degrees of degradation.

2 RELATED WORK

When a task does not receive enough CPU time[1], the duration needed for the task to complete its execution will inadvertently be extended. This extension causes subsequent tasks to be further delayed, a cascading effect that results in an overall lowering of the periodic rate of execution. In [6, 9], the user can determine the progress of the tasks by controlling the allocation of CPU cycles or the duration of the allocation interval. In the event of overloads, degradation is achieved by reducing the allocation of CPU to less important tasks, thus lowering their rate of execution. This is a coarse-grain approach to degradation as quite often, not all portions of a task are equally important and we are not able to simply rank all the tasks in a static order of importance.

[14] takes a probabilistic approach by providing performance guarantees if the computation times of the tasks followed a probability density function. We feel that this assumption may not always be true, especially if the tasks are executed on a non-real time operating system. The use of probabilities may not be the answer to managing overloads for multimedia as it does not guarantee stability in execution and in degenerate cases, may lead to jerky and undesirable output.

[12] studies the algorithmic methods taken by many researchers to manage degradation. They classify the algorithms under three different dimensions – static/dynamic dispatch, critical (importance) aware/unaware, and overload aware/unaware; and performed several simulations under varying conditions. The results show that contrary to popular belief that more information will beget a better schedule, none of the algorithms display a consistent advantage over the others. This reinforces our approach to address overload scheduling from a structural point of view, rather than simply relying on algorithms, which have limited scope.

Imprecise computation technique [10, 1] partitions each task into two sub-tasks, a mandatory sub-task which must be executed at every cycle and a optional sub-task which can be discarded under overloaded conditions. The mandatory sub-task is first executed to obtain an acceptable result while the optional sub-task may be executed subsequently to refine and improve on the result. Of particular interest is the C-type optional sub-task which has an added condition that executing the optional sub-task becomes mandatory if it had been skipped for a certain period of time.

[1] CPU time given to a task is quantified by a specified amount of CPU cycles allocated at fixed intervals

3 THE MULTIMEDIA TASK MODEL

Our proposed task model (see Figure 2) has a hierarchical structure that is made up of a few task sets at the top-most level. Each task set, τ, is a col-

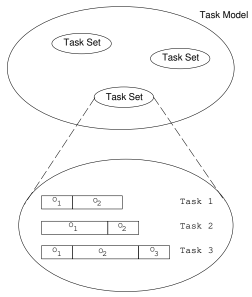

Figure 2: The multimedia task model

lection of several independent tasks, $\{T_1, T_2, \ldots, T_n\}$, where n is the number of tasks in the task set. Each task, in turn, is defined as $T_i = (r_i, p_i, d_i, O_i)$ where r_i is the ready time, p_i is the period and d_i is the relative deadline. The task is further partitioned into a sequence of logical operations (or subroutines), $O_i = [O_i^1, O_i^2, \ldots, O_i^m]$, and each operation O_i^j is a 2-tuple (oc_i^j, f_i^k), where oc_i^j is the computation time of the operation and f_i^k is the *frequency* of the operation. The task model represents the computer system, with each task set as a process. Each process is made up of several threads of execution (tasks), each can be viewed as a media stream in a multimedia application. The operations are logical subroutines which make up the functionality of the task. For example, a simple task will consist of three operations, namely *input* $\rightarrow$ *process* $\rightarrow$ *output*. This is different from treating each operation as a separate task because the sequence of operations (of a task) share a deadline and respect and/or precedence constraints amongst each other. The partitioning is performed by the user, depending on the level of granuality he prefers and the number of operations is not restricted. One important characteristic is that each operation is *non-preemptible*. In multimedia applications, many routines (especially I/O and network) cannot be preempted as they may make use of specialized hardware, or shared data (to reduce large data movements). As such, we treat each operation as atomic and preemption by a higher-priority task happens only at the completion of the operation of the executing task. With this notion, the use of semaphores, which adds execution overheads, for critical sections becomes redundant. Analysis of the model is also more accurate since all preemption points are known in advance and can be accounted for.

To cope with overloads, some form of compromise is needed to reduce the demand on the CPU. Some of the considerations we have are:

- *Task Criticality.* A task often has to perform a series of actions and not all of these actions are equally important. This awareness is highlighted also in imprecise scheduling technique [10, 1]. The task should be as important as the most critical operation it performs, but the other non-critical operations within the task should not share the same level of service.

- *Operation Importance.* Non-crucial operations, when discarded too often, may become critical. In a video-conferencing application, the audio playback is normally classified higher in priority than the video playback. However, if the users do not see regular image updates on their screen, then the whole idea of video-conferencing is defeated.

- *Variable Rate of Execution.* Multimedia applications are periodic in nature, but do not require a strict rate of execution. The degree of variability varies among the different media. With audio data, jitter requirements are much more stringent compared to video data.

- *Uniform Degradation.* In experiments conducted, [4] show that the dropping of frames, while necessary when overloaded, needs to be regulated to ensure smoothness and quality, and we should not wait until the frame is late before discarding it.

To accommodate the considerations, we assign a *frequency* parameter, $f \in [0, 1]$, to each operation. The frequency f represents the acceptable level of execution of the operation with respect to its original rate. The basis of this parameter is similar to the C-type sub-task in imprecise computation technique. An operation with $f = 1.0$ is mandatory, one with $f = 0.0$ is optional and one with $f = 0.5$ means that it has to be executed at every other period. When the computer system cannot provide the application with all the CPU time that it requires, a lower service level is requested and all the operations within the application will degrade according to their frequencies. In this arrangement, degradation within each task can be explicitly controlled with respect to the importance each operation plays in the application. Operations which are more sensitive to periodicity requirements can thus be assigned larger frequency values. The value in f is not taken as a probability but a ratio of the original execution rate which the operation must maintain. In short, in a task T_i with period p_i, the operation, O_i^j with a frequency f_i^j is expected to execute approximately once every $\frac{1}{f_i^j} * p_i$ seconds. This ensures that all operations gets service at a regular rate, rather than bursty instances during overloads.

4 SCHEDULING ANALYSIS OF MODEL

In the task model, if every task is made up of only one operation, it simply translates to a non-preemptive system; and if each operation executes in unit time, then preemption can occur at every instant, resulting in a fully-preemptive system. Informally, this means that the scheduability conditions governing our proposed multimedia task model lies somewhere between the fully-preemptive and non-preemptive model. If we define a feasibility set $F_M = \{\tau \ : \ \tau \text{ is scheduable in task model, } M\}$ which contains all the task sets which are scheduable under each particular model, we have:

$$F_{\text{Non-preemptive}} \subseteq F_{\text{taskmodel}} \subseteq F_{\text{Fully-preemptive}}$$

We can obtain a more accurate bound (omitted here) for the task model by taking into account the largest blocking time which higher priority tasks will face during execution (which is the size of the largest operation) and modifying the proof found in [7].

We are, however, more concerned with how to improve the feasibility bounds to cope with overloads, through the use of the frequency parameter, f, associated with each operation. Rather than trying to tailor a scheduling algorithm to cater to the frequency parameter (which, if not properly designed, may have degenerate cases), we introduce an external filtering process which all operation invocations will undergo before reaching the processor queue. This filtering process, performed by the frequency filter (FF), will decide if that particular operation should be executed or discarded (see Figure 3). If the operation passes through the filter, it is then sent to the processor queue for execution; else, it is rejected and the task may in-

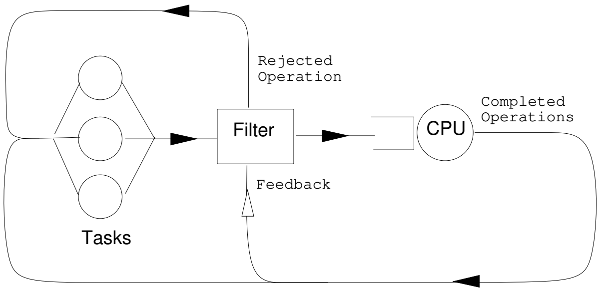

Figure 3: Role of Frequency Filter

voke the next operation. Scheduling at the processor queue can be based on existing scheduling algorithms like EDF or RM [11]. The frequency filter works by keeping track of the rate of execution of the operations and prevents operations with lower frequencies from entering the processor queue too often. At the filter, the algorithm we used is a simple but effective one. Called *Cumulative Filtering*, it maintains a small counter for each operation, and each time an operation is invoked, its frequency is added to its counter. If the value in the counter is larger than or equal to 1, the operation passes through and the counter decreases by 1; else, the operation is discarded. It is trivial to see that the algorithm will ensure that each operation passes through at its stated frequency and also degrades the operation in a uniform manner. This algorithm runs in $O(1)$ time. To minimize demand, we examine a way to spread out the execution of the operations so that not all the operations get accepted or rejected at the same time. For example, if a task, T_i has two operations, O_i^1 and O_i^2, both with a frequency $f = 0.5$, the best possible way to reduce worst case CPU demand on the system is to adjust the execution of the operations such that only one operation gets executed at each invocation (see Figure 4). This

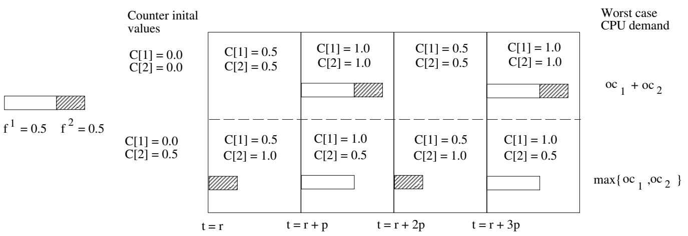

Figure 4: Minimizing worst-case CPU demand

can be achieved by selecting suitable initial values to the counters at the beginning of execution. In the example in Figure 4, if O_i^1's counter has a starting value of 0, and O_i^2's counter has a starting value of 0.5, we get the the spreading out effect on the operations. To generalize this to all other frequency combinations, we show a lemma below:

Lemma 3: For two operations $O_i^1 = (oc_i^1, f_i^1)$ and $O_i^2 = (oc_i^2, f_i^2)$, the worst case CPU demand is $\max\{oc_i^1, oc_i^2\}$ if

$$f_i^1 + f_i^2 = 1.0$$

Proof: We define a value k where $k \leq 0.5$, and $f_i^1 = 1.0 - k$, $f_i^2 = k$. We also place specific values into the counters, where $C[1] = 0.0$ and $C[2] = 1.0 - k$, associated with the operations respectively. Following the execution,

$$\begin{array}{lll} t = 0, & C[1] = 0.0 & C[2] = 1.0 - k \\ t = 1, & C[1] = 1.0 - k & C[2] = 1.0 \geq 1.0 \\ t = 2, & C[1] = 2.0 - 2k \geq 1.0 & C[2] = k \end{array}$$

Hence, at $t = a$,

$$\begin{aligned} C[1] &= [(a-1) - (a-1)k]_{\text{mod } 1.0} + 1 - k \\ C[2] &= [(a-1)k]_{\text{mod } 1.0} + k \end{aligned}$$

We have to prove that at any point of execution, if O_i^1 passes through the filter, then O_i^2 is rejected and vice versa.

$$\begin{aligned} C[1] &= [(a-1) - (a-1)k]_{\text{mod } 1.0} + 1 - k \\ &= 1 - [(a-1)k]_{\text{mod } 1.0} + 1 - k \\ &= 2 - \left([(a-1)k]_{\text{mod } 1.0} + k\right) \\ &= 2 - C[2] \end{aligned}$$

If O_i^1 passes through at $t = a$, then

$$\begin{array}{llll} & C[1] & \geq & 1.0 \\ \Rightarrow & 2 - C[2] & \geq & 1.0 \\ \Rightarrow & C[2] & < & 1.0 \quad O_i^2 \text{ gets rejected} \end{array}$$

If O_i^2 passes through at $t = a$, then

$$\begin{array}{llll} & C[2] & \geq & 1.0 \\ \Rightarrow & 2 - C[2] & < & 1.0 \\ \Rightarrow & C[1] & < & 1.0 \quad O_i^1 \text{ gets rejected} \end{array}$$

Hence, Lemma 3 is proved.

With this lemma, we can then reduce the worst case CPU demand by systematically grouping operations together, and assigning appropriate initial values to their counters at the frequency filter. In addition, we observe that for two frequencies, f_i^1 and f_i^2, if f_i^2 divides f_i^1 where $\frac{f_i^1}{f_i^2} = q$, then it is possible to adjust the invocations of q operations with frequencies

f_i^2 such that the invocations match exactly the invocation of one operation with frequency f_i^1. This observation helps us to extend Lemma 3 to more than two operations in a group. Some possible groupings for operations (with frequencies up to one decimal place) are:

$[(0.1),(0.9)]$,$[(0.1),(0.3),(0.3),(0.3)]$
$[(0.2),(0.8)]$,$[(0.2),(0.4),(0.4)]$,$[(0.2),(0.2),(0.2),(0.4)]$
$[(0.3),(0.7)]$,$[(0.1),(0.1),(0.1),(0.7)]$
$[(0.4),(0.6)]$,$[(0.1),(0.1),(0.2),(0.6)]$,$[(0.2),(0.2),(0.6)]$
$[(0.5),(0.5)]$,$[(0.1),(0.1),(0.1),(0.1),(0.1),(0.5)]$

An inherent limitation in the groupings is that the tasks of all the operations within a group must be synchronous (same starting time) and have the same period. This is reasonable because most multimedia streams are synchronized and are typically executed at a common rate. Since the maximum CPU demand by each grouping is equivalent to the operation with the largest computation time within the group, the overall worst case CPU demand is computed by summing up the CPU demand of all the groups. If the number of groups < number of operations, then we can safely claim that we have overcome the $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$ barrier.

5 BEST EFFORT SCHEDULING

Multimedia applications are generally soft real-time and users are willing to tolerate intermittent synchronization discrepancies and missed deadlines. Many researchers have concurred that optimal scheduling algorithms (EDF,RM,LS) perform badly under overload situations, though Rate Monotonic, by assigning a static priority, is more predictable [12] in most conditions. The problem here is that it is insufficient to simply rely on some priority semantics to degrade the execution. we show here that the use of the frequency parameter in the operations can provide more stable and effective execution under overloaded conditions. As actual execution is nondeterministic and difficult to model accurately, we use a practical approach to demonstrate the results.

A test application is built on a SGI Indy platform (running IRIX 5.3). The application is a periodic video-capture-display task consisting of four operations: (1) *CaptureVideo*, is a mandatory operation which periodically reads a video image from the Indycam; (2) *DisplayVideo* outputs the image onto a window; (3) *EdgeDetection* performs a simple edge-detection routine on the capture image and outputs a trace of the edges onto another window; and (4) *DisplayNurbs* manipulates the video image by texture-mapping it on a spinning NURBS and outputs the b-spline on another window. The task is set to execute at 10 fps and since both *EdgeDetection* and *DisplayNurbs* operations are computationally expensive and highly variable, we can observe the the effects of scheduling under overloaded and nondeterministic conditions. Two experiments are conducted here. The first is a static priority-based system where *CaptureVideo* will receive the highest priority, followed by *DisplayVideo*, *EdgeDetection*, and *DisplayNurbs*. In the second experiment, we follow the setup shown in Figure 3. Since no initial frequency values are assigned, we included an additional routine in the filter to obtain feedback from the scheduler and dynamically reduce the frequencies proportionally (based on their importance) if the scheduler cannot keep up with the current rate.

The graphs compare the performances based on execution adhering to the deadline as well as execution which is late. In Figure 5, we see that while execution

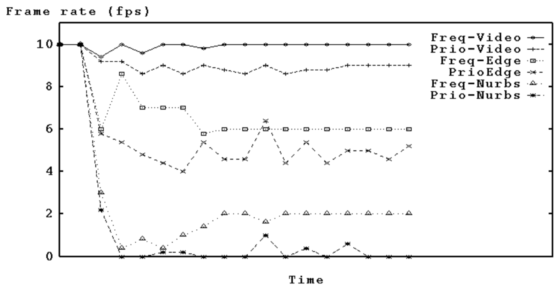

Figure 5: Rate of operations adhering to the deadline

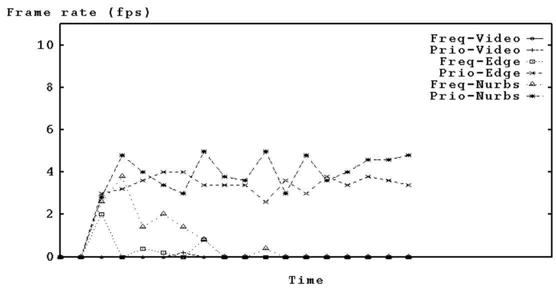

Figure 6: Rate of operations missing the deadline

was spread out proportionally for both the priority-based and frequency-based experiment, the operations scheduled with frequency managed to obtain higher and more stable rates. The initial instability in the execution is attributed to the adjustments made by the filter to find suitable frequencies to assign to the operations. On the other hand, in Figure 6, we see that the experiment using only priorities could not stabilize and thus wasted precious CPU cycles scheduling operations which could not complete by the deadline.

6 CONCLUSION

In this paper, we introduce a task model which represents a multimedia application, broken down into simple, non-preemptible operations. Scheduling during overload is achieved through the use of frequency values assigned to the operations and an external filter which rejects operations which are non-critical and allows only critical operations to reach the scheduler. We also discuss a way to reduce worst case CPU demand by finding possible groupings for the operations and allocating them appropriate start values. We observed the effects of overload scheduling when execution times of the operations are highly variable. Two experiments were conducted, the first using only a static priority-based system while the second used the filtering mechanism and dynamically adjusted the frequencies when necessary. The results showed that the second experiment, with the filter, obtained higher and more stable execution rates.

This scheduling model is part of an on-going project on creating a computer-participative multimedia development and execution environment on UNIX. The task model utilizes an in-house developed user-threads package to achieve both control and flexibility in the scheduling. This threads package, in turn, relies on the real-time extensions in modern UNIX operating systems (e.g. IRIX, Solaris, HP-UX) to provide for more deterministic execution.

References

[1] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1173, September 1990.

[2] L. Delgrossi, C. Halstrick, D. Hehmann, R. Hertwich, O. Krone, J. Sandvoss, and C. Vogt. Media scaling for audiovisual communication with the heidelberg transport system. In *Proceedings of ACM Multimedia 93*, pages 99–104, 1993.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness.* W.H. Freeman and Company, San Francisco, California, 1979.

[4] Richard Gerber and Ladan Gharai. Experiments with digital video playback. Technical Report UMIACS TR 95-103, Department of Computer Science, University of Maryland, October 1995.

[5] B. B. Hehmann, M. G. Salmony, and H. J. Stuttgen. Transport services for multimedia applications on broadband networks. *Computer Communications*, 13(4):197–203, May 1990.

[6] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. *Lecture Notes in Computer Science*, 1018:64–78, 1995.

[7] K. Jeffay, D. F. Stanat, and C. U. Mattel. On non-preemptive scheduling of periodic and sporadic tasks. In R. Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1991*, pages 129–139. IEEE Computer Society Press, December 1991.

[8] H. Kaneko and J. Stankovic. A multimedia server on the spring real-time system. Technical Report UMass CS 96-11, Department of Computer Science, University of Massachusetts, Amherst, 1996.

[9] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. *In the proceedings of Multimedia Japan 96*, April 1996.

[10] K. J. Lin, S. Natarajan, and J. W. S. Liu. Scheduling real-time, periodic jobs using imprecise results. In *Proc. IEEE Real-Time System Symp.*, 1987.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20, pages 46–61. 1973.

[12] M. Marucheck and J. K. Strosnider. An evaluation of the graceful degradation propertiess of real-time schedulers. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, June 1995.

[13] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, MIT Laboratory for Computer Science, 1983.

[14] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, L.-C. Wu J. Sun, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 164–173. IEEE, May 1995.