System Synthesis of Synchronous Multimedia Applications

Gang Qu, Malena Mesarina, and Miodrag Potkonjak
Computer Science Department, University of California, Los Angeles, CA 90095
{gangqu, malena, miodrag}@cs.ucla.edu

Abstract

Modern system design is being increasingly driven by applications such as multimedia and wireless sensing and communications, which all have intrinsic quality of service (QoS) requirements, such as throughput, error-rate, and resolution. One of the most crucial QoS guarantees that the system has to provide is the timing constraints among the interacting media (synchronization) and within each media (latency). We have developed the first framework for systems design with timing QoS guarantees, latency and synchronization. In particular, we address how to design system-on-chip with minimal silicon area to meet timing constraints. We propose the two-phase design methodology. In the first phase, we select an architecture which facilitates the needs of synchronous low latency applications well. In the second phase, for a given processor configuration, we use our new scheduler in such a way that storage requirements are minimized. We have develop scheduling algorithms that solve the problem optimally for a-priori specified applications. The algorithms have been implemented and their effectiveness demonstrated on a set of simulated MPEG streams from popular movies.

1 Introduction

Multimedia applications have intrinsic requirements on deadlines to process the incoming data (latency), and coherent playout of different types of data (e.g. synchronization among text, image, audio, and video or multiple video/audio streams). The timing relationship among the interacting media (synchronization) and within each media (latency)[1] is one of the most important metrics for the quality of service (QoS) provided by the system that supports such applications, and must be satisfied at the presentation[2] time. For example, the lip-sync of audio and video usually requires 25 or 30 synchronization points per second. Cen et al.[1] provide the lip synchronization in a MPEG player by simultaneously displaying audio and video frames with the same sequence number. Qiao and Nahrstedt[7] design a fine-grain lip-sync algorithm that first estimates the audio playback and the video decoding times and then adopts a selective dropping policy for each type of I,P, or B frames. Synchronization has been discussed in both of the recently proposed multimedia standards[3, 6].

Systems design traditionally focuses on the optimization of objectives such as power, cost, area, performance. As embedded CPU cores become increasingly popular in VLSI systems and multiple embedded cores have been integrated on a single silicon, system designers have to implement systems using real-time design techniques to meet the design constraints. Memory hierarchies, in particular caches and on-chip memory, play a very important role in achieving high performance in modern RISC embedded cores. One may use a fast CPU and large caches to improve the performance, however, this requires a large silicon area and restricts the on-chip memory on a fixed silicon area. Research in the context of real-time scheduling suggests that a proper scheduler with certain knowledge of the upcoming applications requires less storage[2, 5]. How to provide QoS guarantees has not received the attention that it deserves in the system design society. In this paper, we address the problem of systems design with these traditional optimization targets and QoS guarantees, in particular, how to take into account synchronization and latency requirements during the system-on-chip (SoC) design.

We propose a two-phase design methodology: (i) selection of hardware configuration and (ii) storage minimization via tasks scheduling. Different processor cores, combined with different sizes of I-cache and D-cache, have different performance. In the phase of hardware configuration selection, we exclude a combination if it does not produce better performance but occupies more area than another one. For each of the remaining hardware configurations, we determine the minimal storage requirement to satisfy the QoS guarantees by finding the optimal scheduling. Then the systems are evaluated and the one with the best performance is chosen based on the optimization targets. We develop an off-line pseduo-polynomial scheduling policy, which is

[1] These are sometimes referred as *intra-media synchronization* and *inter-media synchronization* respectively.

[2] By *presentation*, we mean the delivery of various media to the user. For example, the display of text and images or the dynamic played-out of audio, video and animations.

provably optimal in minimizing the storage under the timing constraints.

A Motivational Example

We use a small example to illustrate the importance of scheduling discipline. Suppose there are two applications, $\mathcal{A}$ and $\mathcal{B}$, to be processed on a single processor. Each application consists of a sequence of tasks that request certain amount of memory storage, CPU time and latency constraints as shown in Table 1.

Arrival Time		0	1	2	3	4	5
Storage Requirement	$\mathcal{A}$	10	2	30	10	1	11
	$\mathcal{B}$	1	20	3	30	10	8
Latency Constraint	$\mathcal{A}$	3	3	3	7	7	7
	$\mathcal{B}$	4	4	4	5	5	5

Table 1. Latency (*in units of CPU time*) and storage requirement (*in units of memory*) for the task sequences of two applications. The i-th task of an application arrives at time i and has a deadline equals to the sum of arrival time and latency.

For simplicity, we assume each task takes exactly 1 unit CPU time for execution. The processor can start executing a task on its arrival and free the memory occupied by this task as soon as it is finished. Tasks in the same application follow the first come first serve (FCFS) strategy. Let $t_{\mathcal{A}_i}, t_{\mathcal{B}_i}$ be the finishing time for the i-th tasks of $\mathcal{A}$ and $\mathcal{B}$. We say $\mathcal{A}$ and $\mathcal{B}$ are *k-synchronized* if $|t_{\mathcal{A}_i} - t_{\mathcal{B}_i}| \leq k$ for all i. We want to schedule the tasks such that no deadline is missed, a pre-defined level of synchronization is achieved and the memory requirement is minimized.

In order to solve the problem, we first construct a storage requirement table (Figure 1), where the entry (i, j) indicates the total storage requirements at the end of time $i + j$ when i CPU units are assigned to $\mathcal{B}$ and j CPU units to $\mathcal{A}$. An entry marked by "X" indicates a situation that at least one of the deadlines is missed. For example, by time 4, from Table 1, we know that tasks $\mathcal{A}_0, \mathcal{A}_1$ and $\mathcal{B}_0$ have to be finished, therefore, any scheduler that reaches entries (0,4), (3,1), or (4,0) will fail to satisfy all the latency constraints.

A *scheduler* is a path from the upper left corner (0,0) to the lower right corner. At any entry, the schedule moves either one step to the right or one step down, and assigns the next CPU time to either $\mathcal{A}$ or $\mathcal{B}$ respectively. The earliest deadline first (EDF) policy[5] always selects the task with the least deadline. In EDF_1, a tie is broken to minimize the number of context switches, in EDF_2, whenever there is a tie, we choose the one that occupies more memory. In this example, both EDF_1 and EDF_2 serve the two applications with a minimal storage requirement of **93** and achieve 3-synchronized as shown by the "solid arrow path" and "dotted arrow path" in Figure 1.

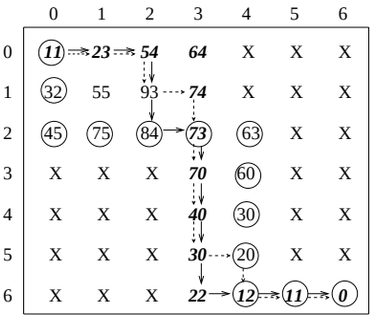

Figure 1. Four possible schedulers ($\rightarrow$: EDF_1, $--\!>$: EDF_2, bold italic font: 3-sync, $\circ$: 2-sync.).

Our off-line optimal algorithm uses dynamic programming to find the minimal storage requirement at any instant time and then finds one scheduler. In this case, an optimal scheduler is the path consisting of entries in the *Bold Italic* font using only **74** memory units and achieves the same synchronization. 2-synchronized is also possible as represented by the circled entries. A comparison of the above 4 schedulers is given in Table 2. One can easily see that scheduling policies can affect the QoS and better synchronization can be achieved at the expenses of extra storage and context switches.

	EDF_1	EDF_2	3-sync	2-sync
storage	93	93	74	84
synchronization	3	3	3	2
# of context switches	4	6	2	5

Table 2. Comparison of the 4 schedulers.

2 Background and Problem Formulation

2.1 Architecture and Hardware Model

Figure 2 shows a typical application specific system-on-chip which consists of microprocessor core(s), instruction cache, data cache, hardware accelerators, control blocks, on-chip memory, etc. Several factors combine to influence the system performance: processor performance, I-cache and D-cache miss rates and miss penalty, and clock speed. In particular, the system performance is computed using the following formula for cycles per instruction (CPI): CPI= $\frac{f}{\text{MIPS}}$+(Miss_Rate$_{\text{I-Cache}}$+Miss_Rate$_{\text{D-Cache}}$) $\times$ Miss_Penalty , where f is the system clock frequency, and MIPS is million instructions per second.

Caches typically found in current embedded multimedia systems range from 4KB to 32KB. Although larger caches corresponds to higher hit rates, they occupy a larger silicon area. Since higher cache associativity results in significantly higher access time, we consider only direct-mapped caches.

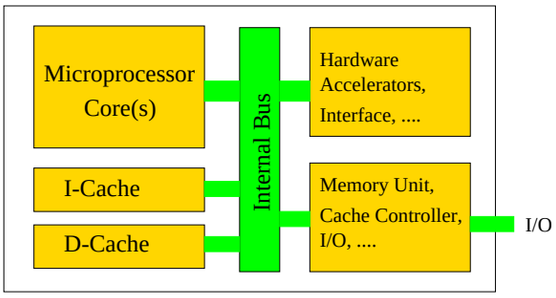

Figure 2. A typical core-based application-specific system-on-chip.

We experimented 2-way set associative caches, but they did not dominate in any single case. Cache line size was a variable in our experimentation. Its variation corresponded to the following trade-off: larger line size results in less hardware and area together with higher cache miss penalty. We use CACTI [9] as a cache delay estimation tool with respect to the main cache parameters: size, associativity, and line size. A sample of the cache model data is given in Table 3.

Cache Size	Cache Line Size				
	8B	16B	64B	128B	512B
4KB	7.6656	7.0208	6.4892	6.5615	-
8KB	8.3447	7.8065	6.9916	6.9057	9.3963
16KB	9.3041	8.5829	7.6205	7.5426	9.7992
32KB	10.4131	9.4499	8.591	8.6902	10.0449

Table 3. Minimal cycle time (ns) for direct-mapped caches with variable line sizes.

Data on microprocessor cores have been extracted from manufacturer's datasheets and the CPU Center Info web [8]. A sample of the collected data is presented in Table 4. The table presents embedded microprocessor core operating frequency, MIPS performance, technology and area. Given a fixed choice of processor core and caches, we can calculate the execution time for a given task. Long execution time implies a large memory to store the tasks that have arrived but not yet been executed.

Processor Core	Clock (MHz)	MIPS	Technology (μm)	Area (mm^2)
ARM7 LPower	27	24	0.6	3.8
LSI TR4101	81	30	0.35	2
LSI CW4001	60	53	0.5	3.5
LSI CW4011	80	120	0.5	7
Motorola 68000	33	16	0.5	4.4
PowerPC403	33	41	0.5	7.5
DSP Group, Oak	80	80	0.6	8.4
NEC, R4100	40	40	0.35	5.4
Toshiba, R3900	50	50	0.6	15
StrongARM	233	266	0.35	4.3

Table 4. The performance and area data for sample processor cores.

2.2 Application and Quality of Service Model

We assume that we receive applications from a reliable end-to-end connection. Each application consists of a set of tasks, each task has its arrival time, latency, execution time (for a given hardware configuration), storage requirement and synchronization specification with the tasks in other applications. Formally, the j-th task $\mathcal{A}_{ij}$ of the i-th application $\mathcal{A}_i$ has the following parameters:

- t_{ij}: the arrival time
- τ_{ij}: the execution time with a given hardware configuration
- l_{ij}: the latest time to finish $\mathcal{A}_{ij}$ after its arrival
- m_{ij}: the memory requirement
- (n_{ij}^k, s_{ij}^k): the synchronization of $\mathcal{A}_{ij}$ and the task in the k-th application, i.e., the finish time of $\mathcal{A}_{ij}$ and $\mathcal{A}_{kn_{ij}^k}$ cannot differ by more than s_{ij}^k unit time[3].

On the service side, we assume that tasks within the same application are processed in the first come first serve (FCFS) fashion, and there is a charge for the context switch among different applications. The memory occupied by a task can be freed as soon as this task have been executed. The execution time for a task depends on the hardware configuration, for example, a fast processor core and large cache with low miss rate provide short execution time.

2.3 Problem Formulation and Key Results

We formulate the problem as follows:

Given a set of applications with their computation, storage, latency and synchronization requirements, determine a system-on-chip (i.e., the type of processor core, sizes of I-cache, D-cache and on-chip memory) with the minimal silicon area such that all the application requirements are satisfied.

We developed a dynamic programming-based algorithm that finds the minimal on-chip storage requirement and a feasible scheduler to service the applications within their timing constraints (latency and synchronization) in pseudo-polynomial time. The algorithm assumes a priori knowledge of the data streams and tasks within the same application are scheduled following the FCFS policy. However, every task can have its individual latency and synchronization requests; we do not assume that computation load is proportional to data size; finally the algorithm is also applicable when a context switch penalty is explicitly specified.

We define a dominance relationship among the possible SOC configurations and select the one that requires minimal silicon area from all the non-dominated configurations. This methodology is valuable in making early design decisions in silicon area allocation among processor, cache, memory and others.

[3]Alternatively, we may use tables or matrices to specify the synchronization among tasks from different applications.

3 Global Synthesis Flow for QoS Guarantees

In this section, we describe the global flow of the proposed synthesis system and explain the function of each subtask and how they are combined into a synthesis system.

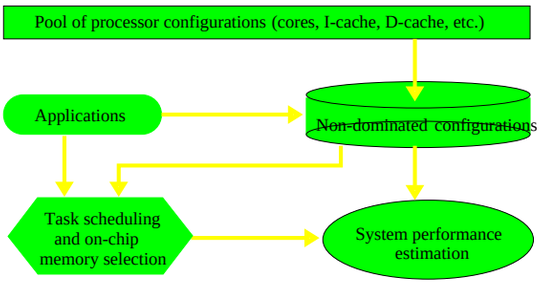

Figure 3. Global flow of the synthesis approach.

The goal is to choose the configuration of processor, I-cache, D-cache and determine a task schedule with minimal storage once the hardware configuration is fixed. To accurately predict the system's performance for target applications, we employ the approach which integrates the optimization, simulation, modeling, and profiling tools. The synthesis technique considers each non-dominated microprocessor core and competitive cache configuration, and selects the hardware setup which requires minimal silicon area and meets all the QoS requirements of the applications.

Figure 3 depicts the global flow of the proposed synthesis approach. Starting from a pool of processor cores, I-cache, and D-cache configurations, we identify all the non-dominated hardware configurations based on the characteristics of the given applications. Then for each such system setup, coupled with the detailed information of the applications, we determine the minimal storage requirement and a task schedule to fulfil the QoS demand. Finally we conduct the system performance estimation, and select the one that optimizes our design goal.

4 Synthesis Techniques

4.1 Resource Allocation

The objective in this phase is to find an area-efficient system configuration since area is our primary optimization target.

We conduct an exhaustive search for all the processor cores, I-cache (range from 512B to 32KB), D-cache (range from 4KB to 32KB) and cache line sizes (from 8B to 512B). For each combination, we estimate the system performance and area. One processor type *dominates* another if it uses less area and results in the same or better system performance. The non-dominated system configurations are kept and task scheduling will be performed on these configurations to identify the most area efficient design.

For each competitive hardware configuration, since the silicon area for storage is proportional to the size of the on-chip memory, our goal is to find the minimal amount of storage that meets the latency and synchronization constraints for a given set of applications. Once the storage requirement is determined, we can do the system performance estimation and in particular calculate the total silicon area.

Finally, a task scheduler is required to schedule the tasks such that neither deadline miss or storage overflow occurs. We argue that this cannot be done unless the hardware configuration is fixed, because the execution time for a task varies with different hardware configurations.

4.2 The Basic Storage Minimization Algorithm

We describe our area minimization algorithm for the simplest case in Figure 4, where we have only two applications $\mathcal{A}_1, \mathcal{A}_2$. Each application has a task arriving at the end of each time unit, requiring 1 unit execution time for the given hardware configuration and m_{ij} unit of storage, further there is no latency and synchronization constraints (i.e., $t_{ij} = j, \tau_{ij} = 1, l_{ij} = \infty,$ and $s_{ij}^k = \infty$ for all $j \geq 0$ and $i = 1, 2.$).

Input: m_{ij}, the storage requirements for $i = 1, 2, 0 \leq j < T$.
Output: M, the minimal memory required and a task schedule with M units of memory.

Procedure:
1. Build the instant memory requirement table IMR:
$IMR_{ij} = IMR_{i-1,j} + m_{1,i+j} + m_{2,i+j} - m_{2,i-1}$
$= IMR_{i,j-1} + m_{1,i+j} + m_{2,i+j} - m_{1,j-1}$ (*)
2. Build the aggregate memory requirement table AMR:
$AMR_{ij} = \max\{IMR_{ij}, \min\{AMR_{i-1,j}, AMR_{i,j-1}\}\}$ (**)
3. Find a path in table AMR from entry (0,0) to (T,T) without crossing any entry that has number larger than AMR_{TT}.
3.1 start from entry (T,T) in table AMR
3.2 while (not reach entry (0,0))
{ mark the current entry
move up or to the left whichever has entry $\leq AMR_{TT}$
}
4. Report this path and $M = AMR_{TT}$.

Figure 4. The dynamic programming-based algorithm for finding the minimal memory requirement and a feasible schedule.

Equation (*) computes the memory requirement *AT* time instant $k = i + j$ when i slots have been assigned to application $\mathcal{A}_2$ and $\mathcal{A}_1$ gets j. This is the amount of total size of tasks that have already arrived but not yet finished, therefore $IMR_{ij} = \sum_{l=j}^{k} m_{1l} + \sum_{l=i}^{k} m_{2l}$ according to the FCFS dicipline. Notice that the instant memory requirement is *path-independent*. That is, i slots and j slots have been as-

signed to applications $\mathcal{A}_2$ and $\mathcal{A}_1$ respectively, but it does not matter to whom each specific slot has been assigned.

Equation (**) finds the minimal memory requirement *UPTO* time instant $k = i + j$. It has to be large enough to store the unfinished tasks ($\geq IMR_{ij}$), and guarantees a feasible path to entry (i, j) from either left ($\geq AMR_{i-1,j}$) or above ($\geq AMR_{i,j-1}$).

In step 3, any marked entry has either its left entry or the entry above or both with value $\leq AMR_{TT}$, which is the minimal storage requirement. This is guaranteed by equation (**). Once the AMR table is built[4], the minimal memory requirement is given as AMR_{TT} and a feasible scheduler (a path from (0,0) to (T,T) in the AMR table) can be found backwards in time $2T$ as in step 3. The complexity of this algorithm is $O(T^2)$ in both time and space.

4.3 Modifications for QoS Guarantees

In this section, we briefly discuss how to modify the above algorithm to meet the QoS guarantees (e.g. latency, synchronization) for general applications (e.g. individual arrival time, latency, execution time) when there is a charge for context switching.

latency: As we have seen in Figure 1, adding (individual) latency constraint simply decreases the amount of computation for building the IMR and AMR tables. For example, if the first task of application $\mathcal{A}_1$ has to be finished by 4, there is no need to compute entries $(i, 0)$ for all $i \geq 4$.

synchronization: Like latency, synchronization constraints reduce the number of entries to be filled in both IMR and AMR tables. For instance, if we want a solution that is 1-synchronized, it will be sufficient to fill only the entries $(i, i), (i - 1, i), (i + 1, i)$.

execution time: Recall that in the IMR table, entry (i, j) is the memory requirement to store the tasks that have arrived but have not been finished yet. So when tasks require different execution time, we only free the storage for the tasks in $\mathcal{A}_1$ that can be finished in j unit time and those in $\mathcal{A}_2$ that can be finished in i unit time.

arrival time: This case is similar to the case when tasks have individual execution time.

N applications: If there are N applications instead of only 2, we have to build a N-dimensional table to find the optimal solution. This of course increase the complexity of the algorithm.

context switch: If there is a charge for the context switching when we make a turn on the path from the upper left corner to the lower right corner. A path (scheduler) with minimal number of turns (context switches) can be found similarly by dynamic programming.

[4]One can easily combine (*) and (**) to build the AMR table directly. Here we use an intermediate table IMR to explain our approach. However, in both cases, the space and time complexity is $O(T^2)$.

5 Experimental Results

We test the proposed algorithms on MPEG video streams. Standard MPEG encoders generate three types of compressed frames: I frames (intra-pictures), P frames (predicted pictures) and B frame (bi-directional predicted pictures). On average, I frames are the largest in size (since they are self-contained), followed by P frames and B frames. Krunz and Tripathi [4] present a comprehensive model for MPEG video streams. In particular, the frame sizes of different types of frames are simulated by three different sub-models which are intermixed according to the group-of-pictures pattern. Statistically, the generated MPEG streams fit the empirical video and are sufficiently accurate in predicting the queueing performance for real video streams. We simulate four video streams using the parameters provided in [4] and the information of the generated frames is reported in Table 5. (The frame size of I-frames has a relatively large standard deviation because it is modelled as the sum of two random components).

Movie	Number of Frames	I-frame size		P-frame size		B-frame size	
		μ_I	σ_I	μ_P	σ_P	μ_B	σ_B
Wizard of Oz	41,700	15.18	13.61	4.82	0.64	3.91	0.27
Star Wars	174,960	8.68	5.51	3.93	0.58	2.81	0.52
Silence of the Lambs	39,972	6.53	2.86	2.59	0.86	1.98	0.70
Goldfinger	40,104	9.77	6.60	4.57	0.51	3.26	0.38

Table 5. Simulation of the MPEG streams (μ. is the mean and σ. is the standard deviation).

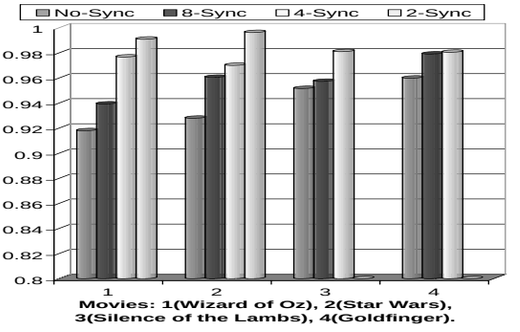

Figure 5. Normalized storage requirement *vs.* synchronization. In movies 3 and 4, 2-synchronized cannot be achieved.

The algorithm in Figure 4 finds the minimal storage requirement for a set of a priori applications. For each of the above MPEG video movies, we find the storage requirements for both the EDF policy and our off-line optimal sched-

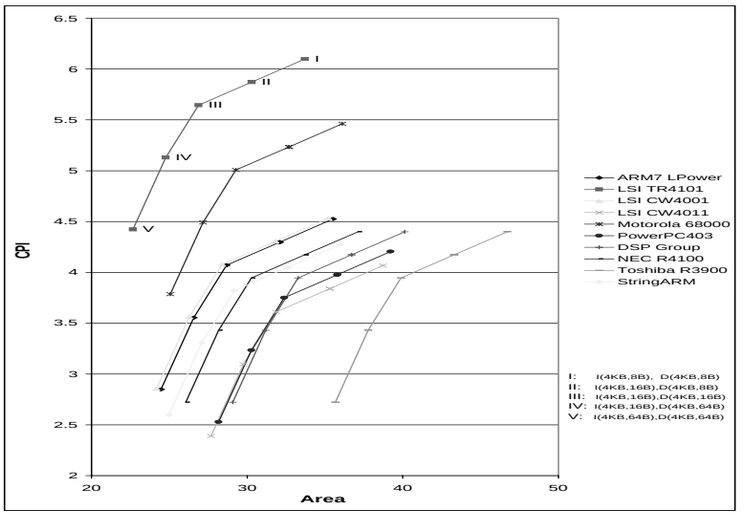

Figure 6. System performance (cycles per instruction) *vs.* **the silicon area (mm^2) for different hardware configurations.**

uler. In EDF, a tie is broken randomly[5]. Our off-line algorithm has been applied four times with no synchronization, 2-sync, 4-sync, and 8-sync. The off-line optimal storage requirements are normalized with respect to that for the EDF policy as shown in Figure 5. The key feature of these solutions is that they have synchronization guarantees and the trend is clear: better synchronization needs more storage. In all cases, the off-line EDF policy, which achieves 12 $\sim$ 15-synchronized, requires more storage.

Different processor cores use different amount of silicon area and deliever different performance (see Table 4). We investigate each processor core with I-cache, D-cache (size varies from 4KB to 32KB) and cache line (size from 8B to 512B) setups. For the sample MPEG frames, we conclude that the LSI TR4101 core with a 4KB I-cache and 4KB D-cache have the best performance in terms of silicon area. Details are reported in Figure 6.

6 Conclusion

In this paper, we address the problem of how to design system-on-chip with minimal silicon area that meets the QoS requirements for real-time applications. We select the timing constraints (synchronization and latency) as the measure for QoS and propose an algorithm to determine the minimal storage and feasible schedule for a given hardware configuration to provide QoS guarantees for given applications. We propose a two-phase design methodology of hardware configuration selection and storage minimization. For a fixed hardware configuration, our storage minimization algorithm provides the optimal solution to meet all the QoS requirements. We show that better synchronization can be achieved at the cost of more storage. Experiments on simulated MPEG movies demonstrate that our scheduler saves storage over off-line EDF policies and provides synchronization guarantees.

[5]Actually, we experiment two EDF policies where the task with the largest memory and the one with least execution time wins the tie respectively. However, they provide solutions of negligible difference in terms of storage requirement.

References

[1] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. *A Distributed Real-Time MPEG Video Audio Player.* Proceedings of the fifth International Workshop on Network and Operating System Support for Digital Audio and Video, pp. 151-162, 1995.

[2] H-J. Chen and T.D.C. Little. *Storage allocation policies for time-dependent multimedia data.* IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.5, pp. 855-864, 1996.

[3] International Organization for Standardization. *Information Technology Coding of Multimedia and Hypermedia Information.* ISO/IEC 13522-1. October 1994.

[4] M. Krunz and S.K. Tripathi. *On the characterization of VBR MPEG streams.* ACM International Conference on surement and Modeling of Computer Systems (SIGMETRICS 97), pp. 192-202, 1997.

[5] C.L. Liu and J.W. Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment.* Journal of ACM. Vol.20, No.1, pp. 47-61, 1973.

[6] B.D. Markey. *HyTime and MHEG.* Digest of Papers, Thirty-Seventh IEEE Computer Society International Conference, pp. 25-40. 1992.

[7] L. Qiao and K. Nahrstedt. *Lip synchronization within an adaptive VOD system.* Proceedings of the International Society for Optical Engineering, pp.170-181, 1997.

[8] http://infopad.eecs.berkeley.edu/CIC/.

[9] S.J.E Wilton and N.P Jouppi. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, Vol.31, No.5, pp. 677-688, 1996.