System Synthesis of Synchronous Multimedia Applications

GANG QU
University of Maryland, College Park
and
MIODRAG POTKONJAK
University of California, Los Angeles

Modern system design is being increasingly driven by applications such as multimedia and wireless sensing and communications, which 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 constraint among the interacting media (synchronization) and within each media (latency). We have developed the first framework for system design with timing QoS guarantees. In particular, we address how to design system-on-chip with minimum silicon area to meet both latency and synchronization constraints. The proposed design methodology consists of two phases: hardware configuration selection and on-chip memory/storage minimization. In the first phase, we use silicon area and system performance as criteria to identify all the competitive hardware configurations (i.e. Pareto points) that facilitate the needs of synchronous applications. In the second phase, we determine the minimum on-chip memory requirement to meet the timing constraints for each Pareto point. An overall system evaluation is conducted to select the best system configuration. We have developed optimal algorithms that schedule a-priori specified applications to meet their synchronization requirements with the minimum size of memory. We have also implemented on-line heuristics for real-time applications. The effectiveness of our algorithms have been demonstrated on a set of simulated MPEG streams from popular movies.

Categories and Subject Descriptors: B.3 [**Hardware**]: MEMORY STRUCTURES; C.3 [**Computer Systems Organization**]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS – Real-time and embedded systems; J.6 [**Computer Applications**]: COMPUTER-AIDED ENGINEERING – Computer-aided design (CAD)

General Terms: Algorithms, Design

Additional Key Words and Phrases: synchronization, on-chip memory minimization, high-level embedded systems synthesis

This work was supported by ...

Authors' addresses: Gang Qu, Electrical and Computer Engineering Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; email: gangqu@glue.umd.edu; Miodrag Potkonjak, Computer Science Department, University of California, Los Angeles, CA 90095; email: miodrag@cs.ucla.edu.

1. INTRODUCTION

Multimedia applications have intrinsic requirements on deadlines to process the incoming data (latency) and the coherent playout of different types of data (e.g. synchronization among text, image, audio, and video or multiple video/audio streams). The timing relationships among the interacting media and within each media are referred as synchronization and latency (or intra-media synchronization and inter-media synchronization in some context) respectively. These synchronization requirements must be satisfied at the time when various media (e.g. the display of text and images, the dynamic played-out of audio, video and animations) is delivered to the user. For example, the lip-sync of audio and video usually requires 25 or 30 synchronization points per second.

Memory structure plays a vital role in the design of embedded multimedia system that must provide such synchronization guarantees. A typical application specific system-on-chip (SoC) consists of a processor core, instruction cache, data cache, background memory, and a set of optional hardware accelerators and control blocks. These components often comprise for more than 70% of SoC area. Memory hierarchy, in particular cache and on-chip memory, is the base for many synchronization-enhancing techniques. It also impacts significantly the embedded system's performance, power dissipation, and overall implementation cost. The synthesis of SoC for synchronous multimedia applications poses several interesting synthesis and optimization problems. For example, data storage and data processing are equally important in multimedia applications. The lack of either storage or processing power may cause violation to the timing constraints. We can use a powerful processor, which usually takes more silicon area, and large cache to provide fast processing speed. However, this limits the on-chip memory for data storage if the total silicon area is fixed. Our goal is to investigate how to balance this trade-off between the processor and the memory such that the synchronization requirements will be satisfied with the minimum silicon area.

In this paper, we lay out a system design framework that simultaneously optimizes traditional design targets (such as area, cost, power, throughput, testability, and scalability) and provides QoS guarantees. We illustrate this by a discussion of, but not limited to, SOC design with minimum silicon area that meets the application's synchronization and latency requirements. We propose a two-phase design methodology: hardware configuration selection and storage minimization. Different processor cores, combined with different configurations of I-cache and D-cache, have different performance. In the phase of hardware configuration selection, we exclude a combination if it requires more silicon area but does not produce better performance. For each of the remaining hardware configurations, we determine the minimum storage requirement to satisfy the QoS guarantees by task scheduling. Then we conduct the system performance evaluation and select the best system configuration based on the optimization targets. We develop both offline and online scheduling policies. Our offline algorithm is provably optimal in minimizing the storage under the timing constraints. The earliest deadline first (EDF) algorithm is one of the most widely used on-line scheduling policies, but it does not give any synchronization guarantees. Experimental results show that our on-line heuristic is competitive with EDF while providing the timing QoS guarantees.

The highlight of this paper is a dynamic programming-based algorithm that finds the minimum storage requirement and a feasible schedule for a single processor to service a set of applications. The algorithm also gives a feasible schedule with this minimum storage. All applications' timing constraints (latency and synchronization) will be satisfied and the algorithm has a pseudo-polynomial run time. The algorithm assumes a-priori knowledge of the data streams. Each data stream consists of a set of tasks which are served by the first come first serve policy. Tasks are allowed to have their individual latency and synchronization requests. The algorithm does not assume that a task's computation requirement is proportional to its storage requirement. Finally the algorithm is applicable when cache miss and context switch penalties are explicitly specified.

The rest of the paper is organized as follows. In the next section, we explain the preliminaries, formulate the problem, and highlight our results. Section 3 shows the detailed storage minimization techniques, where we explain our dynamic programming based optimal scheduling policy, discuss how it provides QoS guarantees with minimum storage requirement, and propose a simple but effective on-line heuristics for real time applications. In Section 4, we present the global SOC design flow for synchronization and other QoS guarantees. We report the experimental results on simulated real media benchmark in Section 5. We discuss the related work in Section 6 before conclude.

2. PROBLEM FORMULATION AND KEY RESULTS

In this section, we first lay out the application model and QoS metrics we use in our approach. We formulate three different but related problems: storage minimization with QoS guarantees, task completion maximization under storage constraints, and silicon area minimization with QoS guarantees. We explain the relevance of these problems and list the highlight of our solutions.

2.1 Application and Quality of Service Model

We assume that a single processor system receives data streams from multiple reliable end-to-end connections. Each data stream is an application that consists of a sequence of tasks. Associated with each task are its arrival time, execution time, latency, storage requirement and synchronization specification with tasks in other applications. Formally speaking, the j-th task $\mathcal{T}_{ij}$ of the i-th application $\mathcal{A}_i$ has the following parameters:

—t_{ij}: the arrival time of task $\mathcal{T}_{ij}$

—τ_{ij}: the execution time of task $\mathcal{T}_{ij}$ with a given hardware configuration

—l_{ij}: latency, i.e., the maximum time that task $\mathcal{T}_{ij}$ can stay in the system. At time $t_{ij} + l_{ij}$, the system will either finish task $\mathcal{T}_{ij}$ or drop it. We define task $\mathcal{T}_{ij}$'s deadline to be $d_{ij} = t_{ij} + l_{ij}$.

—m_{ij}: the memory requirement to storage task $\mathcal{T}_{ij}$ in the system

—(n_{ij}^k, s_{ij}^k): the synchronization of task $\mathcal{T}_{ij}$ and application $\mathcal{A}_k$. $\mathcal{T}_{ij}$ must be synchronized with the n_{ij}^k-th task $\mathcal{T}_{kn_{ij}^k}$ of $\mathcal{A}_k$. Furthermore, let x and y be the finish time for tasks $\mathcal{T}_{ij}$ and $\mathcal{T}_{kn_{ij}^k}$, then the synchronization requirement will be met if and only if $|x - y| \leq s_{ij}^k$.

The pair (n_{ij}^k, s_{ij}^k) gives the QoS requirement on synchronization. Synchronization is only applicable to tasks from different applications as tasks within the same application will be served by the first come first serve policy. To express how a task $\mathcal{T}_{ij}$ from application $\mathcal{A}_i$ should be synchronized with tasks from other applications, for each application $\mathcal{A}_k$, we need to specify which task (if any) needs to be synchronized with $\mathcal{T}_{ij}$ and how tight is the synchronization. In our representation, integers n_{ij}^k is the index of the task from $\mathcal{A}_k$ to be synchronized and s_{ij}^k gives the level of the synchronization. For example, considering the fifth task $\mathcal{T}_{1,5}$ of application $\mathcal{A}_1$, if we have $n_{1,5}^2 = 6$ and $s_{1,5}^2 = 4$, then we know that tasks $\mathcal{T}_{1,5}$ and $\mathcal{T}_{2,6}$ must be synchronized that their finish time cannot be differ by more than 4 unit time.

We mention that there are other alternatives to represent the synchronization requirement among tasks such as tables or matrices. However, the applications' synchronization requirement must be self-consistent as stated in the following synchronization assumption:

Assumption 2.1.1 : (Synchronization assumption) The synchronization requirements of all applications are self-consistent. Suppose that task $\mathcal{T}_{ij}$ needs to be synchronized with task $\mathcal{T}_{kl}$, where $l = n_{ij}^k$, from application $\mathcal{A}_k$ by s_{ij}^k unit time, then we have:

(i) symmetry:. $n_{kl}^i = j, s_{ij}^k = s_{kl}^i$

(ii) transitivity:. $n_{ij}^p = n_{kl}^p, s_{kl}^p \leq s_{ij}^p + s_{ij}^k$ for any applications $\mathcal{A}_p$ and $\mathcal{A}_k$.

The *symmetry* property enforces that any pair of tasks to be synchronized must have symmetric synchronization requirements. In the above example, we have imposed $n_{1,5}^2 = 6$ and $s_{1,5}^2 = 4$ onto task $\mathcal{T}_{1,5}$, which imply that tasks $\mathcal{T}_{1,5}$ and $\mathcal{T}_{2,6}$ are to be synchronized. Therefore we need to associate the following symmetry constraint to task $\mathcal{T}_{2,6}$: $n_{2,6}^1 = 5$ and $s_{2,6}^1 = s_{1,5}^2 = 4$.

The *transitivity* property enforces the triangular inequality on the level of synchronization among any three (or more) tasks. If we add the synchronization requirement between task $\mathcal{T}_{1,5}$ and application $\mathcal{A}_3$: $n_{1,5}^3 = 7$ and $s_{1,5}^3 = 8$, then since both tasks $\mathcal{T}_{2,6}$ and $\mathcal{T}_{3,7}$ have to be synchronized with the same task $\mathcal{T}_{1,5}$, their finish time cannot be arbitrary. The transitivity assumption captures this by requiring $\mathcal{T}_{2,6}$ and $\mathcal{T}_{3,7}$ to be synchronized and sufficiently tight ($s_{3,7}^2 \leq s_{1,5}^2 + s_{1,5}^3 = 4 + 8 = 12$).

The above synchronization assumption, although we make it stronger than necessary for simplicity, is a necessary condition for the applications to be schedulable. It can be relaxed and still make the synchronization requirements self-consistent. In the above example, it is not mandatory to have $n_{3,7}^2 = n_{1,5}^2$. Task $\mathcal{T}_{3,7}$ can be synchronized with task other than $\mathcal{T}_{2,6}$ in application $\mathcal{A}_2$ as long as the completion time of tasks $\mathcal{T}_{1,5}, \mathcal{T}_{2,6}, \mathcal{T}_{3,7}$, and $\mathcal{T}_{2,n_{3,7}^2}$ meet their respective synchronization constraints.

We take the following general assumption on the system's service model:

Assumption 2.1.2 : (Service assumption) The single processor system can start servicing a task as soon as it arrives and can free the memory occupied by this task once it has been served. Tasks in the same application are served in the first come first serve (FCFS) fashion. Pre-emption is allowed and we neglect the overhead for context switch between different applications.

2.2 Problem Formulation

Our goal is to illustrate the design methodology that optimizes traditional design targets and provides QoS guarantees. We choose the size of storage (on-chip memory) and silicon area as design optimization objectives and synchronization and task completion as QoS metrics. In particular, we formulate and study the following three problems:

Problem 2.2.1 : (Storage minimization with QoS guarantees)
Given N applications, find the minimum storage requirement for a single processor system such that the system can schedule all tasks in the applications and finish them under service assumption 2.1.2 and satisfy all the latency and synchronization constraints that meet the synchronization assumption 2.1.1.

Problem 2.2.2 : (Task completion maximization with limited storage)
Given N applications and a single processor system with a fixed amount of storage, find an on-line scheduler to maximize the number of completed tasks under the latency and synchronization constraints.

Problem 2.2.3 : (SoC design with minimum silicon area for QoS guarantees)
Given N applications, design a system-on-chip (i.e., the type of processor core, configuration of I-cache, D-cache and on-chip memory) with the minimum silicon area to provide guarantees to the applications' QoS requirements.

According to the knowledge about the incoming data streams (applications), we can classify problem 2.2.1 as (i) problem with **full-knowledge** where all the information of the applications is a-priori. For instance, offline applications or periodic data streams with samples of one period. (ii) problem with **partial knowledge** where certain statistical information of the data stream is given. And (iii) problem with **zero knowledge** where nothing is known before the stream actually arrives. It is clear that less storage will be required as more information of the applications is known. However, without the complete knowledge of the incoming data streams, it is unavoidable to drop tasks due to missed deadlines, violation of synchronization constraints or lack of storage. The dropped tasks will not meet their timing constraints and this implies that solutions may only exist for problem 2.2.1 with full-knowledge. We restrict our discussion to this case and aim to find the minimum storage and a (offline) scheduler.

When full-knowledge is not available, we relax problem 2.2.1 in the following sense: can we guarantee the completion of all tasks regardless of memory requirement (or assuming an infinitely amount of storage is available)? This becomes the classical (on-line) scheduling problem. Even in this case, Baruah et al. [1994] show that in general, any on-line scheduling algorithm can perform arbitrarily worse (in terms of the number of task drops) than an offline scheduler. Hence, no absolute guarantees of the timing constraints (latency, synchronization, etc.) can be expected. This leads us to the introducing of the second QoS metrics, the task completion rate. We consider a given system processing real time applications in problem 2.2.2 which seeks for an on-line scheduler to minimize the number of dropped tasks.

Finally, we formulate the processor vs. memory trade-off in problem 2.2.3. As we have discussed earlier, the processor configuration and on-chip memory compete for silicon area. Our goal here is to build the smallest chip to deliver QoS guarantees.

2.3 Key results:

We study the above problems in this paper and our key results are as follows:

(1) We solve problem 2.2.1 optimally when the full-knowledge of the applications is available. Our dynamic programming based algorithm finds the minimum storage requirement to provide the QoS guarantees as well as a task scheduler to achieve such guarantees with this minimum storage requirement. When the number of applications is fixed, both runtime and space complexity of the algorithm are polynomial to the number of tasks in each application.

(2) For problem 2.2.2, we propose a parametric approach, where the distribution of the tasks is known and the on-line scheduler estimates the parameters of the distribution based on the history and schedules tasks assuming that the history will repeat in the future. Simulation shows that our on-line scheduler can completely avoid task drops due to overflow with less than 10% extra storage compared to the optimal solution.

(3) We propose a two-phase design methodology for problem 2.2.3: selection of hardware configuration and storage minimization via task scheduling. In the first phase, we investigate the combinations of different processor cores, I-cache, and D-cache setups. We define a dominance relationship among the possible SoC configurations and exclude the dominated combinations. In the second phase, for each non-dominated configuration (Pareto points), we use our solutions to problems 2.2.1 and 2.2.2 to calculate the minimum storage requirement. Then the overall system performance is evaluated and the best configuration (with the smallest silicon area) will be selected.

3. SCHEDULING TECHNIQUES FOR SYNTHESIS

In this section, we first present an optimal algorithm to solve problem 2.2.1 with full-knowledge. The requirement of a-priori information of all the applications can be met under several real life circumstances such as offline applications and on-line periodic applications. We describe our approach in a simplified case and then discuss how this basic algorithm can be modified to handle the general case. We show the complete algorithm by a small example of two applications. We also present an on-line heuristic for problem 2.2.2.

3.1 The Basic Algorithm for Storage Minimization

We describe our area minimization algorithm for the simplest case in Figure 1, 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. Furthermore, 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 $i = 1, 2.$). Then the problem is equivalent to choosing T units of time from a total of $2T$, assigning them to one application and the rest T units to the other (where T is the number of tasks in each application). There are $\binom{2T}{T}$ choices and we want to find one that requires the least amount of storage. The runtime of an exhaustive search will be $O(4^T)$. The dynamic programming based algorithm we develop has both runtime and space complexity $O(T^2)$.

The proposed algorithm has three steps: in step 1, a $T \times T$ instant memory requirement (IMR) table is built whose (i, j) entry contains the storage requirement at time $i + j$ when i and j slots have been assigned to $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. In step 2, the $T \times T$ aggregate memory requirement (AMR) table is built based on the IMR table, where the (i, j) entry contains the storage requirement such that a path from entry $(0, 0)$ to (i, j) is guaranteed with such amount of storage. Notice that the value of entry (T, T) is the minimum storage we require. In step 3, a feasible scheduler (A scheduler is defined as a path from the upper left corner to the lower right corner in the IMR/AMR table, where the only legal moves are moving down or moving to the right.) is found backwards from entry (T, T) to $(0, 0)$.

Input: m_{ij}, the storage requirements for $i = 1, 2, 0 \leq j < T$.
Output: M, the minimum memory required and a schedule with M.
Procedure:
1. Build the instant memory requirement table IMR:
$$IMR_{ij} = IMR_{i-1,j} + m_{1,i+j} + m_{2,i+j} - m_{2,i-1}$$
$$= IMR_{i,j-1} + m_{1,i+j} + m_{2,i+j} - m_{1,j-1} \quad (*)$$
2. Build the aggregate memory requirement table AMR:
$$AMR_{ij} = \max\{IMR_{ij}, \min\{AMR_{i-1,j}, AMR_{i,j-1}\}\} \quad (**)$$
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 $M = AMR_{TT}$.

Fig. 1. The dynamic programming-based algorithm for finding the minimum memory requirement and a feasible schedule.

Correctness of the algorithm:

Lemma 3.1.1 : (the IMR table) Equation (*) builds the IMR table recursively.
[Proof:] AT time instant $k = i + j$, the system has received tasks from both $\mathcal{A}_1$ and $\mathcal{A}_2$ whose arrival time is $\leq k$. Since the system uses i slots for $\mathcal{A}_2$ and J for $\mathcal{A}_1$, the memory occupied by the first j and i tasks from $\mathcal{A}_1$ and $\mathcal{A}_2$ can be freed according to the service assumption 2.2.2. Therefore, the entry (i, j) of the IMR table should have value of the total size of tasks that have already arrived but not yet finished, which is:

$$\begin{aligned} IMR_{i,j} &= \sum_{l=0}^{k}(m_{1,l} + m_{2,l}) - \sum_{l=0}^{j-1} m_{1,l} - \sum_{l=0}^{i-1} m_{2,l} \\ &= \sum_{l=j}^{k} m_{1,l} + \sum_{l=i}^{k} m_{2,l} \\ &= \sum_{l=j}^{k-1} m_{1,l} + \sum_{l=i-1}^{k-1} m_{2,l} + m_{1,k} + m_{2,k} - m_{2,i-1} \end{aligned}$$

$$= IMR_{i-1,j} + m_{1,i+j} + m_{2,i+j} - m_{2,i-1}$$
$$= IMR_{i,j-1} + m_{1,i+j} + m_{2,i+j} - m_{1,j-1}$$

□

Notice that the instant memory requirement is *path-independent*. That is, i slots and j slots have been assigned to applications $\mathcal{A}_2$ and $\mathcal{A}_1$ respectively, but it does not matter to whom each specific slot has been assigned, i.e., which path the scheduler follows to reach entry (i, j) from $(0, 0)$.

Lemma 3.1.2 : (the AMR table) Equation (**) finds the minimum memory requirement *UPTO* time instant $k = i + j$.

[Proof:] The value $AMR_{i,j}$ in the entry (i, j) of the AMR table guarantees (i) there is no overflow at time $k = i + j$, (ii) there exists a path from entry $(0, 0)$ to (i, j) without crossing any entry with value larger than $AMR_{i,j}$.

From (i), $AMR_{i,j}$ has to be large enough to store the unfinished tasks, $AMR_{i,j} \geq IMR_{i,j}$.

Any path from entry $(0, 0)$ to (i, j) has to visit either entry $(i - 1, j)$ or entry $(i, j - 1)$. To guarantee a feasible path, we must have $AMR_{i,j} \geq AMR_{i-1,j}$ or $AMR_{i,j} \geq AMR_{i,j-1}$.

Therefore, a lower bound for the value $AMR_{i,j}$ is

$$\max\{IMR_{i,j}, \min\{AMR_{i-1,j}, AMR_{i,j-1}\}\}$$

□

Theorem 3.1.3 : The algorithm in Figure 1 determines both the minimum storage requirement, which is the value in the lower right corner of the AMR table, and a feasible scheduler.

[Proof:] From Lemma 3.1.2, a storage in the amount of AMR_{TT} is necessary. We now show that it is also sufficient by finding a scheduler requiring memory no more than this amount.

In step 3.2 of the algorithm, starting from the lower right corner (T, T), we can move either up or to the left, whichever has entry with value $\leq AMR_{T,T}$. This is guaranteed by equation (**). Suppose now we are at entry (i, j), also from equation (**) we have either $AMR_{i-1,j} \leq AMR_{i,j}$ of $AMR_{i,j-1} \leq AMR_{i,j}$ (or both). Thus at any time we are able to move upwards or left to an entry with value no larger than the current value.

Define $f(i, j) = i + j$ for all $0 \leq i, j \leq T$. In particular, we have $f(0, 0) = 0, f(T, T) = 2T$, and any move in step 3.2 will decrease the value of $f(.,.)$ at current entry by exactly 1. From above, we know that a move is always possible from any entry except $(0, 0)$. Hence, after $2T$ moves, we will reach $(0, 0)$ and this gives us a path from entry $(0, 0)$ to (T, T). From the construction of this path, it is clear that no entry on the path has value $> AMR_{T,T}$.

□

Complexity of the algorithm:

Corollary 3.1.4 : The algorithm in Figure 1 has both time and space complexity $O(T^2)$.

[Proof:] Clearly from Lemma 3.1.1 and Lemma 3.1.2, $O(T^2)$ time and $O(T^2)$ space are required to build the IMR/AMR tables (Actually, one can easily combine equations (*) and (**) to build the AMR table directly. Here we use the intermediate

IMR table to explain our approach. However, this does not change the complexity of the algorithm.). Once the AMR is built, the minimum storage requirement is known as $AMR_{T,T}$, and finding a path needs $2T$ runtime as specified in Theorem 3.1.3. □

3.2 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:

Adding individual latency constraint for each task decreases the amount of computation for building the IMR and AMR tables. From the arrival time and latency we have defined the deadline for a task (as the sum of arrival time and latency). When we build the IMR and AMR tables, whenever we detect that an entry may violate the deadline constraint of any task, there is no need to compute the value for this entry and we simply put a special mark on it. For example, if the first task of application $\mathcal{A}_1$ has to be finished by 4, then we mark the entries $(i, 0)$ for all $i \geq 4$ since the deadline is missed if we reach these entries.

synchronization:

Let $f_{1,i}, f_{2,i}$ be the finish time for the i-th task in application $\mathcal{A}_1$ and $\mathcal{A}_2$ as in Section 3.1, we say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are *k-synchronized* if

$$|f_{1,i} - f_{2,i}| \leq k$$

holds for all i. Like latency, synchronization constraints reduce the number of entries to be filled in both IMR and AMR tables. For example, if we want a 1-synchronized solution for the problem in Section 3.1, it will be sufficient to fill the entries $(i, i), (i-1, i), (i+1, i)$, since any of the other entries corresponds to state where synchronization is violated.

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 need 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. If pre-emption is not allowed, the index of the tables can be changed to the finish time of a task and may not be consecutive. In this case, the size of the table (and hence the complexity of the algorithm) will be $O(N_1 \cdot N_2)$ where N_i is the number of tasks in application $\mathcal{A}_i$.

arrival time:

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

context switch:

There will be a charge for context switching when we make a turn on the path from the upper left corner to the lower right corner. A path with minimum number of turns can be found similarly by dynamic programming.

From the above discussion, we immediately have:

Theorem 3.2.1 : The algorithm in Figure 1 can be modified to solve the problem when each individual task has its arrival time, execution time and requires latency

and synchronization. Also a scheduler with minimum number of context switch can be found. Moreover, the complexity of the algorithm will not increase.

N applications:

If there are N applications instead of only 2, we have to build N-dimensional tables to find the optimal solution. For example, if we have three applications, we can extend equation (**) as follows and build the 3-dimensional AMR table:

$$AMR_{ijk} = \max\{IMR_{ijk}, \min\{AMR_{i-1,j,k}, AMR_{i,j-1,k}, AMR_{i,j,k-1}\}\}$$

This of course will not change the correctness of the algorithm but will increase the complexity. In particular, if the i-th application has n_i tasks and requires k_i units of time, the complexity will be $O(\prod_{i=1}^{N} k_i)$ in the pre-emption case and $O(\prod_{i=1}^{N} n_i)$ when pre-emption is not allowed.

3.3 A 2-Application Example

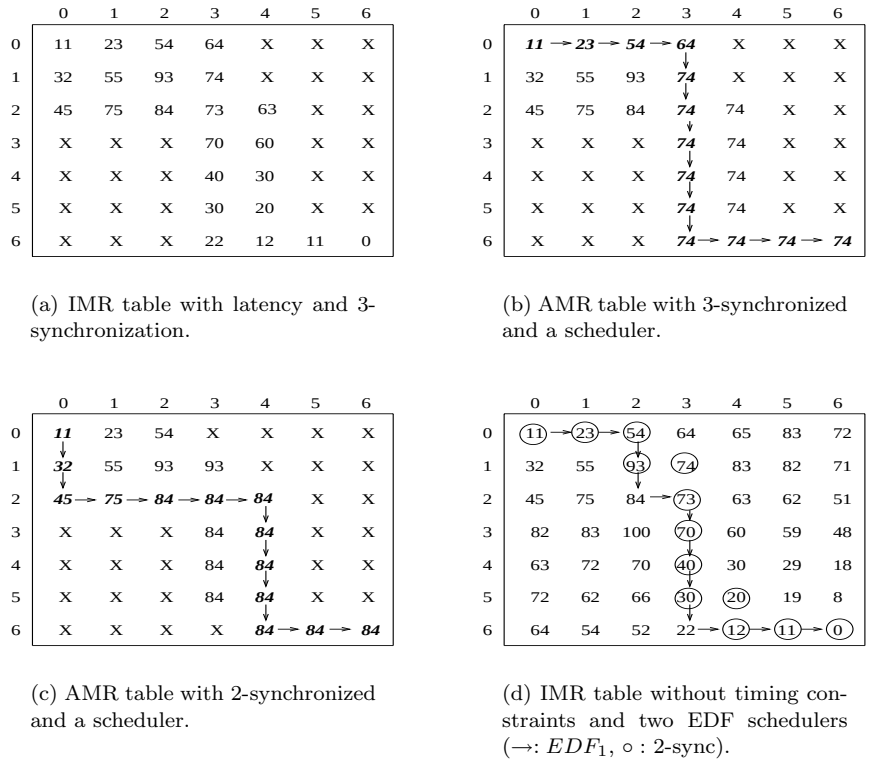
We use a small example to illustrate the complete algorithm and compare it to the widely used earliest deadline first (EDF) scheduling policy. Suppose that there are two applications, $\mathcal{A}$ and $\mathcal{B}$, to be processed on a single processor. Each application consists of a sequence of tasks that request certain amount of memory storage, CPU time and latency constraints as shown in Table I. For simplicity, we assume each task takes exactly 1 unit CPU time for execution and tasks $\mathcal{A}_i$ and $\mathcal{B}_i$ need to be synchronized as good as possible (please refer the paragraph on synchronization in Section 3.2 for the definition of *k-synchronized*).

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
Deadline	$\mathcal{A}$	3	4	5	10	11	12
	$\mathcal{B}$	4	5	7	8	9	10

Table I. Latency (*unit CPU time*) and storage requirement (*unit memory*) for the task sequences of 2 applications. The i-th task of each application arrives at time unit i. The deadline is the sum of arrival time and latency.

We first construct the instant memory requirement (IMR) table (Figure 2(a)), 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 $\mathcal{A}$. The table is filled row-by-row and left to right by the recursive equation (*). For example, if we give one of the first 3 CPU units to $\mathcal{B}$ and the rest to $\mathcal{A}$, entry (1,2) will be filled by *(aggregate memory request for both $\mathcal{A}$ and $\mathcal{B}$ by time 3) - (memory assigned to the tasks that are finished by time 3) = [(10+1)+(2+20)+(30+3)+(10+3 0)] - (1+10+2) = 93*. An entry marked by "X" indicates a situation in which at least one of the deadlines is missed. Consider for example at time 4, from Table I, we know that tasks $\mathcal{A}_0, \mathcal{A}_1$ and $\mathcal{B}_0$ have to be finished by this time, therefore, any scheduler that reaches entries (0,4), (3,1), or (4,0) will fail to satisfy all the latency constraints. Then the aggregate memory requirement (AMR) table is built from

equation (**) and the lower right corner indicates the minimum storage requirement to satisfy all these timing constraints (Figure 2(b)).



(a) IMR table with latency and 3-synchronization.

(b) AMR table with 3-synchronized and a scheduler.

(c) AMR table with 2-synchronized and a scheduler.

(d) IMR table without timing constraints and two EDF schedulers (→: EDF_1, ∘ : 2-sync).

Fig. 2. IMR and AMR tables and four possible schedulers.

A *schedule* 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. In Figure 2(b), a schedule is shown which meets the deadlines of all tasks and achieves 3-synchronized. There are only two context switches: at the start of slot 4, switching from application $\mathcal{A}$ to $\mathcal{B}$ and then switch back when $\mathcal{B}$ is finished at slot 10. Figure 2(c) shows that 2-synchronized is also possible at the cost of more storage and context switches.

The earliest deadline first (EDF) policy [Liu and Layland 1973] always selects the task with the least deadline. We schedule $\mathcal{A}$ and $\mathcal{B}$ by EDF with different tie-break strategies. (A tie is the situation when there are two or more tasks have the same 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 minimum storage requirement of **93** and achieve 3-synchronized as shown in Figure 2(d). A comparison of the above 5 schedulers is given in Table II.

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

Table II. A comparison of the storage requirement, level of synchronization, and the number of context switch for four schedulers.

Our offline optimal scheduler (Figure 2(b)) results in a path requiring only **74** memory units and **2** context switches but achieves the same level of synchronization as both EDF schedulers. 2-synchronized is also possible as shown in Figure 2(c) at the cost of **10** more memory units over the 3-synchronized solution. However, it is still better than both EDF schedulers. One can easily see from Table II that scheduling policies can affect the QoS and better synchronization can be achieved at the expenses of extra storage and context switches.

3.4 On-Line Heuristics for Real Time Applications

The dynamic programming-based algorithm above requires a-priori knowledge of all the applications. The computation also becomes expensive as the number of applications increases. Therefore, developing on-line schedulers becomes a necessity particularly for real-time applications. However, due to the uncertainty of the arriving tasks and the scheduler's on-line nature, it becomes theoretically impossible to provide any kind of timing guarantees. For example, Baruah et al. [1994] show that in the general case, even without considering the storage, any on-line scheduling algorithm can perform arbitrarily worse (in terms of the number of task drops) than an offline scheduler.

Our goal here is to develop on-line scheduling policies such that the expected task drops will be acceptable with reasonable sized on-chip storage. In the proposed heuristic, the processor collects information (frame size and execution time) from each application, predicts their statistical behavior and schedules current tasks based on that the history will repeat. Apparently, the key challenge is how to estimate the data stream's statistical behavior accurately, which, to the great extend, determines the effectiveness of our on-line scheduler. Fortunately, there have already been many discussions on the characteristics of MPEG video streams [Heyman et al. 1994; Bavier et al. 1998; Jabbari et al. 1993; Krunz et al. 1995; Lazar et al. 1993; Reininger et al. 1994; Krunz and Tripathi 1997]. We use the models proposed by Krunz and Tripathi [1997] and by Bavier et al. [1998] to predict the frame size and execution time for the variable-bit-rate MPEG video streams.

Empirical studies by Krunz and Tripathi [1997] indicate that the scene i) the scene length distribution can be appropriately fitted by an exponential (or geometric) distribution, ii) the size of an I frame can be modeled by a sum of two random components: a scene-related component and an AR(2) component that accounts for the fluctuations within a scene, and iii) the sizes of P and B frames can be characterized by two lognormal distributions with different parameters. Bavier et al. [1998] suggest to predict the execution of MPEG frames based on frame size and frame type. They find that it is possible to construct a linear model of MPEG decode time with R^2 values of 0.97 or higher. The corresponding prediction is

accurate to within 25% of the actual decode time despite the great variability of MPEG decoding time.

Another important component of the on-line scheduler is the drop policy, i.e., which task(s) the system will drop if there is not sufficient memory for all the tasks (*overflow* occurs). For each MPEG frame $\mathcal{T}$, we assign a *weight* upon its arrival by the following formula:

$$w(\mathcal{T}) = k - \frac{m}{M} - \frac{e}{E} + \alpha_I + \alpha_P \qquad (***)$$

where k is the number of frames that $\mathcal{T}$ needs to synchronize with, m and e are $\mathcal{T}$'s frame size and execution (decode) time, M and E are the size and decode time for an average sized frame of the same type as $\mathcal{T}$, $\alpha_I > \alpha_P > 0$ are two constants and will be added to $w(\mathcal{T})$ if and only if frame $\mathcal{T}$ is an I frame or a P frame respectively. This comes from the following observations: i) if a task is dropped, all tasks that have to be synchronized with it will be forced to drop as well; ii) I frame is the most important frame in MPEG video stream. An I frame drop will make the decoding of its trailing P/B frames incorrect until the next I frame. A P frame drop will also affect the decoding of its neighbor B frames. α_I and α_P are two parameters that measure how many frames will be affected by the drop of an I/P frame. iii) in case of overflow, the best way to free memory and to keep the task drop rate low is to drop tasks that occupy large memory or require long execution time.

Procedure: on-line task scheduler for fixed size storage.

1. **while** (there exist unscheduled or new arrived tasks or on the completion of a task)
2. { check current time and drop tasks that have missed and will miss their deadlines;
3. assign weights to the new tasks by the following formula (***);
4. check storage and drop least important tasks in case of overflow;
5. update the parameters of the corresponding frame models on a new arrival;
6. schedule the next task based on the current tasks and the predicted future;
7. }

Fig. 3. The pseudo-code of the on-line task scheduler.

Figure 3 outlines the proposed on-line scheduler for a system with limited storage. It will be executed on the completion of a task or on the arrival of a new task until all the tasks are scheduled. In step 2, we drop obsolete tasks and tasks that have no chance to be completed (those who need longer execution time than the remaining time between the current time and their deadlines). In step 3, we assign weights to the new arrivals based on formula (***). This weight will not be changed during the task's entire lifetime. Whenever overflow occurs (or is predicted), tasks with the smallest weight will be dropped first in step 4. We apply a parametric method to predict the size and execution time of future frames using the distributions given by Krunz and Tripathi [1997] and Bavier et al. [1998]. On each new arrival, we get its size and execution time. Then verify and update the parameters for the corresponding model as shown in step 5. Finally, the scheduler selects one task to execute in step 6. The decision is made from the following information: (i) the timing constraints of all the unscheduled tasks, (ii) the occupied storage, and

(iii) the prediction that the upcoming tasks will have the average size and require average execution time according to the most updated statistical models.

4. SYNTHESIS FOR QOS GUARANTEED SOC DESIGN

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 4 depicts the global flow of the proposed synthesis approach. For a set of given applications with their QoS requirements, our goal is to select a processor core, configure the I/D cache, and determine the size of on-chip memory to provide QoS guarantees. 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 that requires minimum silicon area and meets all the QoS requirements of the applications.

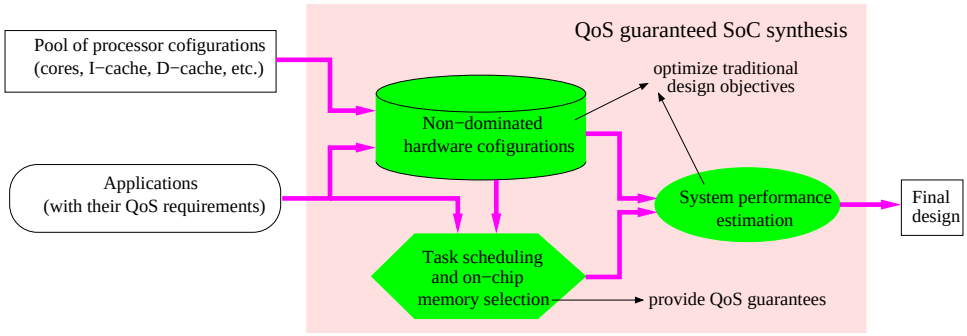


Fig. 4. Global system synthesis flow for guaranteed QoS.

Starting with a library of processor cores, I-cache, and D-cache configurations, we identify all the non-dominated hardware configurations (Pareto points of performance and other design goals such as power and silicon area) 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 minimum storage requirement and a task scheduler to fulfill the QoS demand. The last step is to conduct an overall system performance estimation and pick the one that optimizes our design goal. For example, we choose the one that uses the smallest silicon area if the size is the primary design optimization target.

A typical modern application specific system-on-chip 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/D cache miss rates and miss penalty, and clock speed. In particular, we compute the system performance by the following formula for cycles

per instruction (CPI):

$$\text{CPI} = \frac{f}{\text{MIPS}} + (\ \text{Miss\_Rate}_{\text{I-Cache}} + \ \text{Miss\_Rate}_{\text{D-Cache}}) \times \ \text{Miss\_Penalty}$$

where f is the system clock frequency, and MIPS is million instructions per second.

Caches found in current embedded multimedia systems range from 4KB to 32KB. Although larger caches correspond to higher hit rates, they occupy a larger silicon area as well, resulting in a design trade-off particularly when chip size is one of the primary design concerns. We consider only direct-mapped caches because higher cache associativity results in significantly higher access time. We experiment 2-way set associative caches, but they do not dominate in a single case. Cache line size is a variable in our experimentation. Its variation corresponds to the following trade-off: larger line size results in less hardware and area together with higher cache miss penalty. We use CACTI [Wilton and Jouppi 1996] 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 III.

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 III. 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 (*http://infopad.eecs.berkeley.edu/CIC/*). A sample of the collected data is presented in Table IV. 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. The execution time impacts the amount of memory needed to service the applications. Long execution time implies large on-chip memory to store the arrived but not executed tasks.

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 IV. The performance and area data for sample processor cores.

The application-driven search for a core-based system requires usage of trace-driven cache simulation for each promising point considered in the design space. We attack this problem by carefully scanning the design space using search algorithms with sharp bounds and by providing powerful algorithmic performance techniques. We use the system performance and simulation platform based on SHADE, DINEROIII and a custom analyzer [Kirovski et al. 1997]. 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 (in terms of CPI). The non-dominated system configurations (Pareto points of performance and area in this case) are kept and task scheduling will be performed on these configurations to identify the most area efficient design. This approach is similar to the one in [Hong et al. 1999] while they search for the power-performance Pareto points.

We measure the chip size by the total silicon area occupied by the processor core, I/D cache, and on-chip memory. In general, high performance system has fast processing speed and thus requires less storage to provide the same QoS guarantees than low performance ones. The silicon area for storage is proportional to the size of the on-chip memory, therefore a dominated system will always consumes more silicon area than the ones that dominate it. Consequently, we only need to consider the non-dominated system configurations. In the hexagon of Figure 4, we apply the task scheduling techniques that we discussed in Section 3 to determine the minimum on-chip memory size for each Pareto point to meet the application's QoS requirement. This has to be done for each different non-dominated hardware configuration because task's execution time varies with different hardware configurations. Once we have determined the storage requirement, the best design can be found by an overall system performance estimation, in particular via the estimation of total silicon area.

5. SIMULATION RESULTS

We use simulated MPEG video streams as the target multimedia application and the microprocessor cores reported in Table IV as the pool for our hardware selection. We first explain the method to simulate the frame information of MPEG video streams. Then for a reference system, we report the memory requirement by our scheduling technique to provide synchronization guarantees. A comparison to the EDF policy shows the effectiveness of our approach. Next we briefly discuss the selection of hardware configurations. Finally we present the results on the proposed on-line heuristics for real time video streams.

5.1 Simulation of MPEG streams

We test the proposed algorithms on MPEG video streams. Table V represents the sizes of the compressed frames of four MPEG-encoded video movies, where "Frames" is the number of total frames in the movie, "I-to-I" and "I-to-P" are the distances of I-to-I frame and I-to-P frame respectively.

Standard MPEG encoders generate three types of compressed frames: I frames (intra-pictures), P frames (predicted pictures) and B frame (bi-directional predicted

Movie	Frames	I-to-I	I-to-P
Wizard of Oz	41,760	15	3
Star Wars	174,066	12	3
Silence of the Lambs	40,000	12	3
Goldfinger	40,000	12	3

Table V. Characteristics of four video movies (redrawn from [Krunz and Tripathi 1997]).

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 [1997] present a comprehensive model for MPEG video streams. This model captures the bit-rate variations at multiple time scales. Long-term variations are captured by incorporating scene changes, which are noticeable in the fluctuations of I frames. In particular, the

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 VI. Simulation of the MPEG streams ($\mu.$ is the mean and $\sigma.$ is the standard deviation).

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 queuing performance for real video streams. From the parameters given in [Krunz and Tripathi 1997] we simulate the above four video movies and the information of the generated frames is reported in Table VI. (The frame size of I-frames has a relatively large standard deviation because it is modeled by the sum of two random components).

5.2 Offline Optimal Scheduler

To demonstrate the proposed scheduling technique's advantages of saving memory and providing synchronization guarantees, we conduct the following experiment for each of the above MPEG video movies: First, we apply our offline optimal scheduler to find the storage requirements under a virtual reference system configuration to achieve various levels of synchronization. This is repeated for four times for no synchronization, 2-sync, 4-sync, and 8-sync respectively. Then, to compare the storage requirements, we implement the earliest deadline first (EDF) scheduling policy under the same reference system. We experiment two EDF policies where a tie is broken by the largest memory task first and the least (remaining) execution time task first respectively. However, they do not give solutions significantly different in terms of storage requirement. The offline optimal storage requirements are normalized with respect to that for the EDF policy as shown in Figure 5.

The EDF policy picks the most urgent task first and a tie is normally broken randomly. Therefore it cannot guarantee any level of synchronization unless the synchronization requirement coincides with the task's deadline. In our simulation,

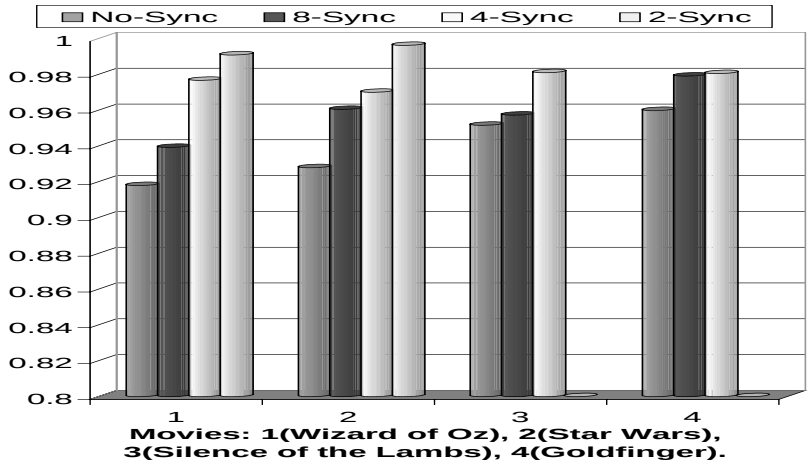


Fig. 5. Normalized storage requirement *vs.* synchronization. In movies 3 and 4, 2-synchronized cannot be achieved.

the solution by EDF is 12 $\sim$ 15-synchronized. Our algorithm, which is based on Figure 1 with simple modification for QoS constraints as discussed in section 3.2, finds the minimum storage requirement for a given hardware configuration. Although the memory saving is not significant, about 6% when no synchronization requirement is specified and less than 4% on average to deliver synchronization, the key contribution of this algorithm is the guaranteed QoS. For each of the above MPEG video movies, our scheduler finds a way to provide services that are 8-, 4-, and 2-synchronized (except in the last two movies, 2-sync schedules do not exist due to the tight timing constraint.). In addition, one can see the clear trend: better synchronization needs more storage. Finally, in all cases and regardless of the synchronization guarantees, our scheduler requires less memory than EDF. This is a coincidence and one can easily find examples where EDF is the most memory efficient scheduling policy.

5.3 Selection of Hardware Configuration

We explain how we determine the non-dominated hardware configurations, i.e., the Pareto points of silicon area and system performance. We measure the silicon area by the size of processor core and the area needed by I/D cache. System's performance is measured by cycles per instruction (CPI) as we discussed in Section 4. We say one hardware configuration (core type and I/D cache setup) dominates another if it achieves at least the same performance with less or same silicon area. It is sufficient to consider only the non-dominated system configurations for the most area efficient design to deliver synchronization guarantees.

We consider 10 different processor cores that are popular for embedded systems. Table IV gives their technology, area, clock frequency and MIPS. For each processor core, we investigate various I/D cache configurations. In our simulation, the size of I-cache and D-cache varies from 4KB to 32KB and cache line size from 8B to 512B.

processor core	LSI TR4101	LSI CW4001	ARM7 LPower	StrongARM	LSI CW4011
area (mm^2)	22.67	24.17	24.47	24.97	27.67
performance (CPI)	4.42	2.86	2.85	2.60	2.39

Table VII. The non-dominated system configurations for 4KB I-cache and 4KB D-cache (cache line size can be varied from 8B to 64B).

We estimate the cache performance, in particular, the area and miss rate by the online cache design tool (*http://arith.stanford.edu/tools/cachetools.html*). For the sample MPEG frames, Table VII lists all the non-dominated system configurations when we fix the size of I-cache and D-cache at 4KB each but allow the cache line size to be changed between 8B and 64B. Figure 6 reports details with five Pareto points connected by the thick line.

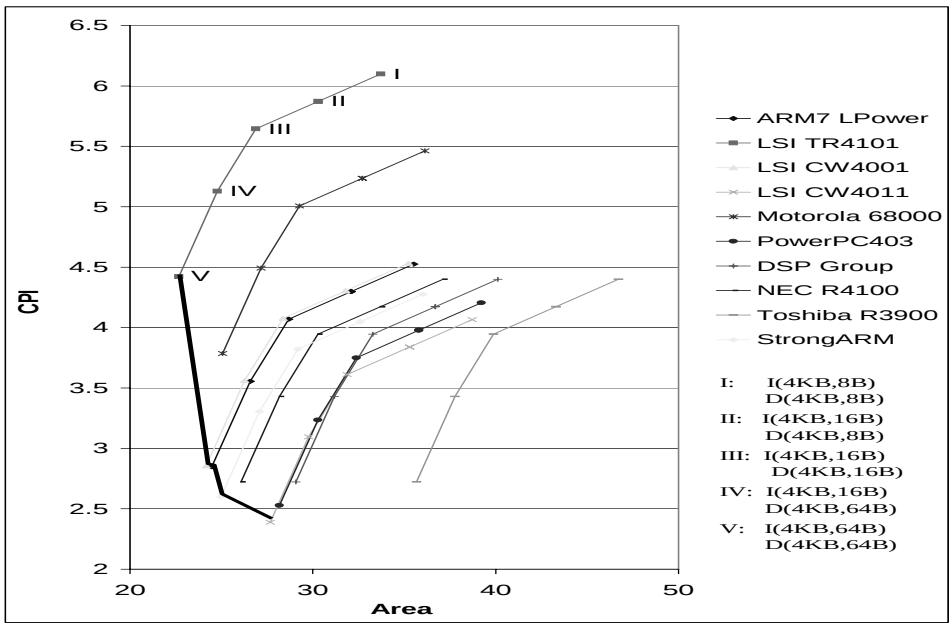


Fig. 6. System performance (cycles per instruction) *vs.* the silicon area (mm^2) for different hardware configurations. The thick solid line connects all five Pareto points.

5.4 On-line Heuristics

We have some general discussion on the design of on-line heuristics in Section 3.4. We implement a simple on-line scheduler based on the pseudo-code in Figure 3. The performance of this on-line algorithm is evaluated by simulation and we report the results in terms of memory size and task drop rate.

We assume that certain statistical information of the frames, such as the I-to-I distance, I-to-P distance, average size and its standard derivation of the each type of frame discussed in Tables V and VI, are known a-priori. Then based on such information, we generate a sample MPEG stream and perform the offline optimal scheduler to determine the (expected) size of storage to satisfy the synchronization requirements. Using the same amount of storage, we apply the on-line heuristics to process the benchmark MPEG video stream at real time. Notice that a frame will be dropped by the on-line scheduler if it is not possible to process the frame before its deadline (adjusted according to its synchronization constraints). Another scenario to drop a frame is when there is not sufficient memory left to store the new arrived data stream. However, it is not always the case to drop the new arrived frame. The drop policy considers each frame's size, execution time, relevance to other frames, and a rough prediction to the next arrival to decide which frame to be dropped. We then increase the size of the on-chip memory and repeat the same simulation. The number of dropped frames and the reason to cause such drops are traced during the simulation. Table VIII reports these results from the simulation on samples of 261 frames for each of the above four MPEG movies.

Movie	Scheduling policy	Normalized Selected Storage			
		1.00	1.05	1.10	1.20
Wizard of Oz	On-line	9.96%	6.90%	1.53%*	1.53%*
	Offline EDF	6.90%	4.21%	0.38%*	0.38%*
Star Wars	On-line	9.56%	7.66%	2.30%*	2.30%*
	Offline EDF	7.66%	4.60%	1.53%*	1.53%*
Silence of the Lambs	On-line	8.42%	3.45%*	3.45%*	3.45%*
	Offline EDF	7.66%	3.45%	3.45%*	3.45%*
Goldfinger	On-line	8.05%	2.68%*	2.68%*	2.68%*
	Offline EDF	6.13%	1.92%*	1.92%*	1.92%*

Table VIII. Frame drop rates on a sample of 261 frames. Storage is normalized with respect to the optimal offline scheduler. *indicates that frame drops are all from missed deadlines.

In Table VIII, we choose the on-chip memory to the same as the one determined by the offline optimal scheduler (which guarantees the completion of all frames), 5% larger, 10% larger, and 20% larger than that size respectively. For each case, we simulate the proposed on-line heuristics and compare its performance, in terms of frame drop ratio, to an offline EDF policy. When we use the same size, the EDF policy sees 7% of the frames being dropped due to the burstiness of the MPEG data stream. The on-line heuristics suffers a frame drop rate at the average of 9%. However, the frame drop rate can be (greatly) reduced by increasing the size of the storage. As the simulation results indicate, the on-line heuristic completely eliminates the frame drops due to the lack of storage and becomes competitive to the offline EDF scheduler with 10% extra storage. If we continue to increase the size of the storage, as shown in the last column, the scheduler cannot further reduce the frame drop. This is because that at this point, all the frame drops are caused by missing the application's timing constraints (either latency or synchronization). Using more storage alone will not help to complete more frames unless we increase

the computing power at the same time. Finally, we mention that our on-line heuristic considers both latency and synchronization constraints, while the offline EDF considers deadline/latency alone and does not give any synchronization guarantees.

6. RELATED WORK

In this section, we survey the previous efforts in the following relevant areas: delivery of synchronization guarantees in multimedia applications, QoS modeling and measurement, system design for QoS, and storage minimization by task scheduling.

The problem of how to deliver such synchronization guarantees has received a lot of attention from communication and multimedia societies. It has been studied as an operating system delivery problem, a physical disk modeling problem, a physical data organization problem, a conceptual database problem, and as a real-time CPU scheduling problem [Anderson et al. 1991; Baqai et al. 1996; Chen and Little 1996; Herman et al. 1998; Liu and Layland 1973; Panzieri and Roccetti 1997]. In particular for MPEG streams, Cen et al. [1995] provide the lip synchronization in a MPEG player by simultaneously displaying audio and video frames with the same sequence number. Qiao and Nahrstedt [1997] 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 is not the only metrics for the quality of service (QoS) of multimedia applications. How to measure the QoS has been a fundamental and challenging problem by itself. The quality of the complex real-time, distributed multimedia services should be application specific, user dependent and thus it is hard to find an explicit one-fit-all definition for QoS. Little and Ghafoor [1990] define the QoS for multimedia communication as a combination of speed ratio, utilization, average delay, maximum jitter, maximum bit error rate, and maximum packet error rate. Lawrence [1997] discusses the metrics based on the QoS attributes of timeliness, precision, and accuracy that can be used for system specification, instrumentation, and evaluation. Kornegay et al. [1999] define QoS of the implementation of an application as a function of the properties of the application and its implementation as observed by the user and/or the environment. Cruz [1995] and Sariowan et al. [1995] introduced the arrival curve and service curve in the context of packet-switch networks. From these curves, one can view QoS in terms of backlog, transmission delay and throughput. The problem of satisfying service guarantees becomes a scheduling problem to meet the backlog and latency constraints. Rajkumar et al. [1997] present an analytical approach for satisfying multiple QoS dimensions in a resource-constraint environment. However, the real time nature of the multimedia applications with human beings as the end users makes synchronization one of the most important QoS metrics and primary design concern of multimedia systems. Evidently, we see that synchronization has been discussed in both of the recently proposed multimedia standards: the MHEG (Coded Representation of Multimedia and Hypermedia Information) and the HyTime (Hypermedia Interchange Standard) [ISO/IEC 13522-1 standard and Markey 1992].

Systems design traditionally focuses on the optimization of objectives such as the minimization of power consumption, area, cost and the maximization of throughput, testability, scalability. How to provider such application specific QoS guar-

antees has not received enough attention that it deserves in the system design society. Kornegay et al. [1999] illustrate the interaction between QoS and synthesis and compilation tasks and discuss the synthesis issues related to the design of QoS-sensitive systems. Qu and Potkonjak [2000] consider the issue of QoS system design with focus on how to minimize energy consumption with the QoS guarantees. The key of their techniques is dynamical voltage scaling. Similar idea has been applied to find the minimum buffer size that maximum the energy saving for multimedia applications [Im et al. 2001] and to minimize energy consumption under a limited sized buffer to deliver timing QoS guarantees on battery-operated systems [Manzak and Chakrabarti 2001].

Several task scheduling approaches have been proposed to take the memory issues into account. Research in the context of real-time scheduling suggests that a proper scheduler with certain knowledge of the upcoming applications requires less storage [Chen and Little 1996; Liu and Layland 1973]. Ade et al. [1994] give upper bounds on the minimum buffer memory requirement for certain synchronous applications. However, their upper bounds are quite loose since they target the minimum buffer memory for all valid schedules and the ones that minimize memory may require much less buffer memory than this upper bound. Murthy and Bhattacharyya [1999] develop a buffer merging technique to reduce data buffering requirement by overlaying buffers in the synchronous dataflow graph. They report a 60% reduction of the buffering memory consumption. Most recently, Maestre et al. [2001] present a general framework for reconfigurable computing, in which task scheduling and context allocation problems have been studied to prune the system design space and to minimize the memory fragmentation.

7. CONCLUSION

In this paper, we address the problem of how to design system-on-chip with minimum silicon area that meets the QoS requirements for real-time multimedia applications. We select the timing constraints (synchronization and latency) as the measure for QoS and propose an algorithm to determine the minimum 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 offline scheduler saves storage over EDF and provides synchronization, the on-line heuristics is effective and efficient in reducing (or completely avoiding) frame drops due to the lack of storage.

REFERENCES

ADE, M., LAUWEREINS, R., AND PEPERSTRAETE, J.A. 1994. Buffer Memory Requirements in DSP Applications. In *Proceedings of IEEE Workshop on Rapid System Prototyping*, June 1994, pp. 108-123.

ANDERSON, D.P. AND HOMSY, G. 1991. A Continuous Media I/O Server and its Synchronization Mechanism. *Computer, Vol.24, No.10*, 51-57.

BAQAI, S., KHAN, M.F., MIAE, W., SHAIKH, S., KHOKHAR, A.A., AND GHAFOOR, A. 1996. Quality-based evaluation of multimedia synchronization protocols for distributed multimedia in-

formation systems. *IEEE Journal on Selected Areas in Communications, Vol.14, No.7*, 1388-1403.

BARUAH, S.K., HARITSA, J., AND SHARMA, N. 1994. On-line scheduling to maximize task completions. In *Proceedings Real-Time Systems Symposium*, 1994, 228-236.

BAVIER, A., MONTZ, B., AND PERTERSON, L. 1998. Predicting MPEG Execution Times. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 98)*, June 1998, 131-140.

CEN, S., PU, C., STAEHLI, R., COWAN, C., AND WALPOLE, J. 1995. A Distributed Real-Time MPEG Video Audio Player. In *Proceedings of the fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1995, 151-162.

CHEN, H-J. AND LITTLE, T.D.C. 1996. Storage allocation policies for time-dependent multimedia data. *IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.5*, 855-864.

CRUZ, R.L. 1995. Quality of Service Guarantees in Virtual Circuit Switched Networks. *IEEE Journal on Selected Areas in Communications, Vol.13, No.6*, 1048-1056.

HERMAN, I., CORREIA, N., DUCE, D.A., DUKE, D.J., REYNOLDS, G.J., AND VAN LOO, J. 1998. A standard model for multimedia synchronization: PREMO synchronization objects. *Multimedia Systems, Vol.6, No.2*, 88-101.

HEYMAN, D., TABATABAI, E., AND LAKSHMAN., T. 1994. Statistical analysis of MPEG2-coded VBR video traffic. In *Proceedings of the Sixth International Workshop on Packet Video*, 1994.

HONG, I., KIROVSKI, D., QU, G., AND POTKONJAK, M. 1999. Power Optimization of Variable Voltage Core-Based Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 12*, 1702-1714.

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

IM, C., KIM, H., and HA S. 2001. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *Proceedings of the 2001 international symposium on Low power electronics and design*, August 2001, pp. 34-39.

JABBARI, B., YEGENGOLU, F., KUO, Y., ZAFAR, S., AND ZHANG, Y.-Q. 1993. Statistical characterization and block-based modeling of motion-adaptive coded video. *IEEE Transactions on Circuits and Systems for Video Technology, Vol. 3, No. 3*, 199-207.

KIROVSKI, D., LEE, C., MANGIONE-SMITH, W., AND POTKONJAK, M. 1997. Application-driven synthesis of core-based systems. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*. San Jose, CA, November 1997, 104-107.

KORNEGAY, K.T., QU, G., AND POTKONJAK, M. 1999. Quality of Service and System Design. In *IEEE Computer Society Annual Workshop on VLSI, Theme: System Level Design*, Orlando, FL, April 1999, 112-117.

KRUNZ, M., SASS, R., AND HUGHES, H. 1995. Statistical characstics and multiplexing of MPEG streams. In *Proceedings of the IEEE/INFOCOM' 95 Conference*, Boston, MA, April 1995, 455-462.

KRUNZ, M. AND TRIPATHI, S.K. 1997. On the characterization of VBR MPEG streams. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97)*, 1997, 192-202.

LAWRENCE, T.F. 1997. The quality of service model and high assurance. In *Proceedings of 1997 High-Assurance Engineering Workshop*, 1997, 38-39.

LAZAR, A.A., PACIFICI, G., AND PENDARAKIS, D.E. 1993. Modeling video sources for real-time scheduling. In *Proceedings of IEEE/GLOBECOM'93, Vol. 2*, 1993, 835-839.

LITTLE, T.D.C. AND GHAFOOR, A. 1990. Network considerations for distributed multimedia object composition and communication. *IEEE Network, Vol.4, No.6*, 32-40,45-49.

LIU, C.L. AND LAYLAND, J.W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM. Vol.20, No.1*, 47-61.

MAESTRE, R., KURDAHI, F.J., FERNANDEZ, M., HERMIDA, R., BAGHERZADEH, N., and SINGH, H. 2001. A framework for reconfigurable computing: task scheduling and context man-

agement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, No. 6*, 858-873.

MANZAK, A. AND CHAKRABARTI, C. 2001. Voltage scaling for energy minimization with QoS constraints. In *Proceedings of 2001 International Conference on Computer Design*, 2001, pp. 438-443.

MARKEY, B.D. 1992. HyTime and MHEG. In *Digest of Papers, Thirty-Seventh IEEE Computer Society International Conference*, 1992, 25-40.

MURTHY, P.K. AND BHATTACHARYYA, S.S. 1999. A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. In *Proceedings of the 12th International Symposium on System Synthesis*, San Jose, CA, November 1999, 78-84.

PANZIERI, F. AND ROCCETTI, M. 1997. Synchronization support and group-membership services for reliable distributed multimedia applications. *Multimedia Systems, Vol.5, No.1*,1-22.

QIAO, L. AND NAHRSTEDT, K. 1997. Lip synchronization within an adaptive VOD system. In *Proceedings of the International Society for Optical Engineering*, 1997, 170-81.

QU, G. AND POTKONJAK, M. 2000. Energy Minimization with Guaranteed Quality of Service. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design*, July 2000, 43-48.

RAJKUMAR, R., LEE, C., LEHOCZKY, J., AND SIEWIOREK, D. 1997. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997, 298-307.

REININGER, D., MELAMED, B., AND RAYCHAUDHURI, D. 1994. Variable bit rate MPEG video: Characteristics, modeling and multiplexing. In *Proceedings of the 14th International Teletraffic Congress,* June 1994, 295-306.

SARIOWAN, H., CRUZ, R.L., AND POLYZOS, G.C. 1995. Scheduling for quality of service guarantees via service curves. In *Proceedings Fourth International Conference on Computer Communications and Networks (ICCCN'95)*, 1995, 512-520.

WILTON, S.J.E. AND JOUPPI, N.P. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits, Vol. 31, No. 5*, 677-688.

Received xxx xxxx; accepted xxx xxxx