# A Tool for Performance Estimation of Networked Embedded End-Systems

*Asawaree Kalavade*
*DSP and VLSI Systems Research Dept.,*
*Bell Labs, Holmdel, NJ 07733.*
*kalavade@bell-labs.com*

*Pratyush Moghé*
*Network and Service Management Research Dept.*
*Bell Labs, Holmdel, NJ 07733.*
*pmoghe@bell-labs.com*

## Abstract

*Networked embedded systems are expected to support adaptive streaming audio/video applications with soft real-time constraints. These systems can be designed in a cost efficient manner only if their architecture exploits the "leads" suggested by clever compile-time performance estimators. However, performance estimation of networked embedded systems is a non-trivial problem. The computational requirements of such systems show statistical variations that stem from several interacting factors. At the slowest time scale, applications can adapt to network bandwidth by configuring the processing functionality of their tasks (e.g. compression parameters). Also, there could be significant execution time variations within a task. Thus, it is tricky to compute the net processing demand of several such applications on a system architecture, especially if the system schedules these applications using prioritized run-time schedulers.*

*In this paper, we describe an analytical tool called AsaP for fast performance estimation of such embedded systems. AsaP builds approximate models of these systems and characterizes the processing load on the system as a stochastic process. The output of AsaP is an exact distribution of the processing delay of each application. This is a powerful result that can be leveraged for efficient design of multimedia networked systems requiring soft real-time guarantees. It is also the first known framework that quantifies the effect of run-time schedulers (FCFS, RM, EDF) on the performance of such systems.*

## 1. Introduction

Examples of networked embedded end-systems include slim hosts like personal digital assistants and network computers [1]. In their next generation, these systems are expected to support multiple concurrent continuous media (CM) applications like audio, video, and graphics. These applications are computation-intensive, periodic, and becoming network-aware. Figure 1 shows two networked embedded systems. The transmitter captures data, encodes it, packetizes it using the particular transport protocol, and sends it over the network. One candidate protocol is RTP (Real-time Transport Protocol [2]), where packets are tagged with timing information. A complementary protocol, RTCP (Real-time Transport Control Protocol), is used by the receiving end to provide feedback regarding packet loss, delay, jitter, etc. This feedback is used by the sender system to *adapt* audio and video applications. If the network is congested, a video application manager adapts the video application to send at a lower frame rate, trading off bandwidth for perceptual quality [3]. When network conditions improve, a higher frame rate may be used. Different frame rates are referred to as *adaptation levels*. The audio application can also have adaptive behavior; one of PCM, ADM, LPC, and GSM compression algorithms can be selected at run time depending on network feedback [4]. These algorithms trade off output bit rates with computation load. For example, the relative processing loads of PCM and LPC is 1:100. This adaptive behavior is one characteristic of "smart networked" embedded systems.

In addition to adaptive behavior, the applications running on these end-systems also tend to have varying performing constraints ranging from relatively strict real-time (e.g. audio) to tolerating looser constraints (e.g video). The heterogeneous demands of these applications are typically managed using sophisticated run-time schedulers on the end-systems.

Our focus in this paper is on the system-level design of such systems. We believe that estimating the performance of a set of adaptive applications on a particular architecture is one of the key problems in system-level design of networked end-systems. Further, the estimation has to account for the particular run-time scheduling policy that may be used to prioritize applications. Performance estimation is difficult since:

1. Each adaptation level offers a different processing load.
2. Data- and architecture-dependent variations (cache misses, bus contention, communication overhead, ...) lead to variable execution times of tasks *within* each application.

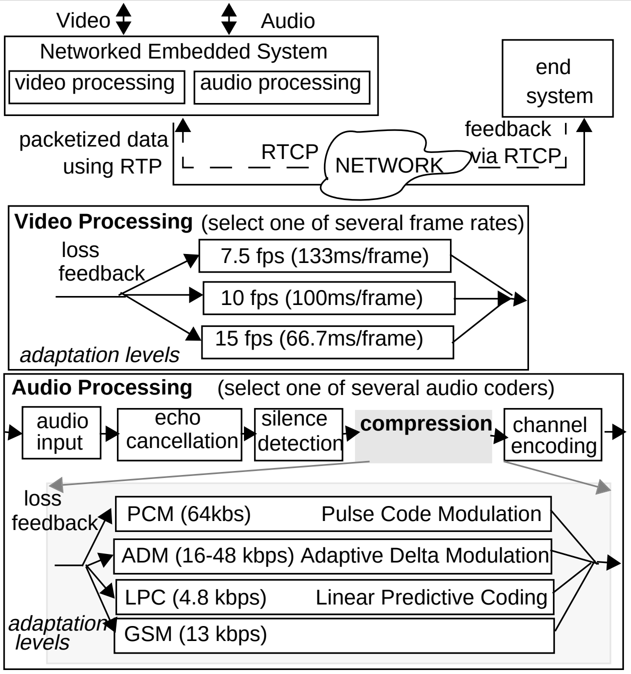

**Figure 1. Networked Embedded Systems. Examples of adaptive applications.**

**35th Design Automation Conference ®**

1-58113-049-x-98/0006/$3.50
DAC98 - 06/98 San Francisco, CA USA

3. The scheduling policy used by the *run-time scheduler* and application priorities impact the processing delay.

The contribution of this paper is to propose an analytical framework to estimate the performance of a set of concurrent adaptive applications on a given architecture and run-time scheduler, wherein the dynamics at any of the above three categories can be captured systematically. The output of performance estimation is several metrics including the **processing delay** (time taken to process one iteration of the application), probability of soft deadline misses, and resource utilization. Our approach is as follows. The first step consists of modeling the adaptation of each application according to network feedback by a time-varying dynamic process called the **adaptation process**. The transitions of this process indicate the probability of jumping from one adaptation level to another and approximate the behavior of the network state and the adaptation algorithm being employed. The second step is to model each adaptive application as a task graph with several alternative execution paths, each path corresponding to a particular adaptation level. Only one of several paths in the task graph is activated at run time, depending on the current adaptation level. This provides the link between the adaptation levels and their processing requirements. Unfortunately, this dependence makes the analysis of the computation complicated. We assume that task-level changes (order of milliseconds) are much faster than changes in the adaptation levels (order of minutes). This assumption makes it much simpler to characterize the state of computation on the end-system between two consecutive changes in the adaptation process. In the third step we characterize the state of computation between changes in the adaptation levels by another process called the **computation process**. The computation process captures dynamics due to the tasks and the run-time scheduler. To characterize this process, we need to capture the task-level resource requirements within each application. We assume that each task in an application has an execution time with an arbitrary known distribution. This distribution captures data-dependent variations as well as other architecture-dependent variations such as cache misses. Communication costs are modeled by adding dummy nodes between tasks. The run-time environment on the end-system is assumed to activate tasks to be run according to a specified scheduling policy. Policies such as earliest deadline first, first-come first served, and rate-monotonic scheduling can be modeled. We show that the computation process is a semi-Markov stochastic process. The exact processing delay distribution of the applications can be derived in terms of the stationary distribution of the jumps made by the computation and adaptation processes. The processing delay distribution can be used to compute the *excess delay probability*, the probability that the delay exceeds a specified threshold. This is a powerful metric for computing soft deadline misses. Other metrics such as average delay and worst-case delay can also be computed.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we propose our analytical framework (*AsaP*). In Section 4, we describe three experiments that demonstrate the use of AsaP for exploring design tradeoffs.

## 2. Related Work

There has been considerable work reported in the literature on estimation of execution times of individual tasks and entire applications in both software and hardware communities. Li *et al*. [5] study the problem of estimating the worst-case execution time of tasks in cached systems using integer linear programming. Gupta *et al.* [6] use stochastic techniques to analyze execution times of single tasks, especially to estimate the execution times of loops with non-deterministic behavior. Henkel *et al*. [7] propose fast simulation-based techniques, for use in hardware/software partitioning, that estimate the execution times for tasks mapped to hardware and software.

Kim *et al* [8] and Li *et al* [9] compute the distribution of processing delay of an application by convolving density functions of individual tasks. Our approach differs in several aspects: First, we support a more general model for tasks; execution times of tasks are not constrained to have an identical distribution [8] but can have an arbitrary distribution. Second, their methods assume a single application; we support concurrent applications under different run-time scheduling policies. Finally, our approach is to model the system-wide behavior through a single stochastic process. The statistics fall out of the characterization; intensive convolutions are not needed.

The work by Yen *et al*. [10] focuses on computing the worst case processing delay of concurrent applications, where each application is made up of tasks and the execution time of a task is specified as a tuple, representing the lower and upper bound. Note that our formulation, in contrast, models the execution time of a task as a *distribution*, which is more powerful in expressing variations in execution time; worst case execution times can be one or two orders of magnitude larger than the actual execution times [8]. We compute the distribution of the processing delay, in addition to the worst-case execution time. We comment more on this in the results section.

In a related area of real-time operating systems (RTOS's), static analysis of RTOS's using deterministic models has been studied extensively [11][12] The RTOS community has also focussed on designing schedulers to give guarantees to multimedia tasks [13][14]. The problem of automated generation of RTOS's has been addressed in the context of hardware/software codesign [15][16][17]. To the best of our knowledge, our paper is the first that analytically explores the combined impact of different scheduling policies, task variations, and network feedback on the effective processing time of concurrent applications using a probabilistic framework.

# 3. Modeling and analysis methodology

## 3.1. Application specification and architecture

We assume that each application is specified by a task graph consisting of nodes representing task-level computations and arcs specifying precedences between nodes (Figure 2). When multiple arcs merge into a node, the node is set to run only when it receives

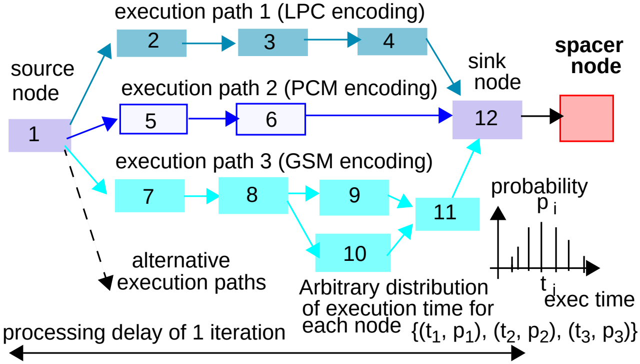
execution path 1 (LPC encoding)
2
3
4
source node
sink node
spacer node
execution path 2 (PCM encoding)
1
5
6
12
execution path 3 (GSM encoding)
7
8
9
11
10
probability
p i
alternative execution paths
Arbitrary distribution of execution time for each node
t i
exec time
{(t1, p1), (t2, p2), (t3, p3)}
processing delay of 1 iteration

**Figure 2. Application task graph with alternative execution paths. Each node has an arbitrarily distributed execution time.**

data from all its incident arcs. Without loss of generality, we assume that the execution time of a node is a discrete random variable that takes one of the values $t_1$, $t_2$, .. $t_k$ with probabilities $p_1$, $p_2$, .. $p_k$ respectively, where these probabilities sum to 1. Such a distribution can approximately model execution time variations due to data and architectural dependencies. Note that the framework can model dependencies in execution time across different nodes; we will not go into details due to lack of space. A final note on distributions: Although they are dependent on the nature of input data, we believe that reasonable distributions can be obtained for specific classes of applications such as video conferencing and action-dominated movies. Analysis using such traffic models has been quite successful for network design.

Due to the repetitive behavior of CM applications, we assume that data samples arrive at the source node at a fixed period dictated by the input rate (e.g. 33ms for 30fps video). We assume non-pipelined execution, i.e., the next iteration of an application starts only after the current iteration finishes. The deadline constraint on each application is assumed to be equal to the period. The actual time when new samples are taken in by the source node for processing depends on when the earlier samples finish processing. This behavior is modeled through the artifice of a dummy **spacer node** that is assumed to run in each iteration after the sink node. The execution time of the spacer node changes based on the elapsed time in each iteration. If the processing delay of an iteration is less than the inter-arrival period, the spacer node models the idle wait of the application until new samples arrive. If the processing delay is more than the period, new samples are made to wait and the execution time of the spacer node is 0.

An application consists of several alternative execution paths corresponding to adaptation levels. To model concurrent applications, task graphs of individual applications are combined into a single *aggregated graph*. Note that the measurement interval for the aggregated graph is *not* the LCM of the periods, which has been traditionally used in literature [12]. Since the execution times of tasks are variable, the LCM does not account for all possible scenarios (also observed by Wolf *et al.* [10]). Instead, we define a *regeneration state*, which is the time when all applications finish execution at the same time. Section 3.3 explains how to compute this. The processing delay is computed over the regeneration interval.

Figure 3 shows the assumed system architecture consisting of programmable processors and hardware accelerators communicating over a user-selected communication fabric. In this work we assume that the number and types of processors and hardware accelerators and the mapping of nodes is known. Each processor runs a RTOS that selects the next task to run. The framework can model different scheduler policies (e.g. rate monotonic, earliest deadline first, ...). Tasks are assumed to be non-preemptive.

## 3.2. Adaptation process

The adaptation process approximates the changes in the adaptation level of applications by an adaptation process. We approximate the possibility of different feedback reports in an adaptation level by different probabilities (Figure 4). Transitions between adaptation levels $m$ and $n$ are assumed to occur with a probability $\nu_{mn}$. This probability is determined by profiling the adaptation of applications [3]. Note that these transition probabilities are time-homogeneous; this may not be a valid approximation in practice, but we believe it is reasonable as a starting point. The interval between jumps corresponds to the interval between consecutive RTCP reports [2] and is modeled as a uniformly distributed random variable. If $M_t^a$ represents the adaptation level of application $a$ at instant $t$, the joint process $M$ = $\{M_t^1, M_t^2, .. M_t^k\}$ represents the joint adaptation level of all the applications currently running on the end-system. Once the transition probabilities $\nu$ are computed, the stationary distribution $\Lambda_m$ is computed by solving the linear equations: $\Sigma\, \Lambda_m\, \nu_{mn} = \Lambda_m$, and $\Sigma\, \Lambda_m = 1$. Due to constant intervals between jumps, the steady-state distribution $\gamma_m$ is equal to $\Lambda_m$. Intuitively, the steady state distribution $\gamma_m$ is the fraction of long-term time spent by the joint adaptation process $M$ in level $m$.

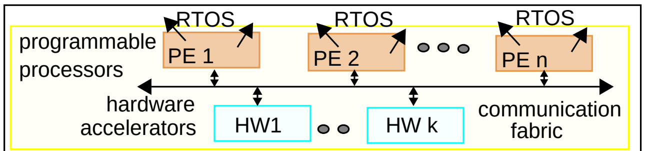

**Figure 3. System architecture.**

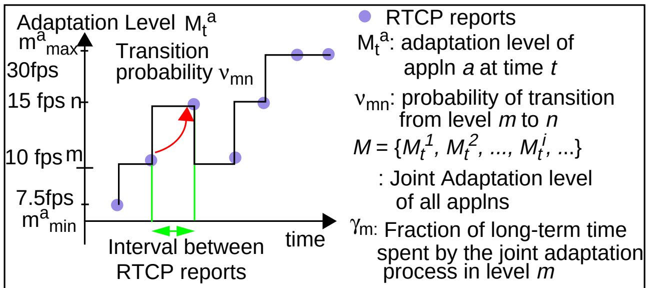
Adaptation Level $M_t^a$
$m^a_{max}$
30fps
15 fps n
10 fps m
7.5fps
$m^a_{min}$
Transition
probability $\nu_{mn}$
Interval between
RTCP reports
time
● RTCP reports
$M_t^a$: adaptation level of appln $a$ at time $t$
$\nu_{mn}$: probability of transition from level $m$ to $n$
$M = \{M_t^1, M_t^2, ..., M_t^i, ...\}$
: Joint Adaptation level of all applns
$\gamma_m$: Fraction of long-term time spent by the joint adaptation process in level $m$

**Figure 4. Adaptation process for application a.**

## 3.3. Computation process

We model the state of computation in the end-system for a particular combination of adaptation levels as a **computation process.** This process is shown to be a *semi-Markov stochastic process*. Such a process has a one-step memory and the stationary probability distribution of its jumps to a particular state can be exactly computed. This in turn enables us to compute the processing delay of an application. Now for the details.

Let us define a few variables that characterize the state of computation as a function of time. Let the transition sequence $A = \{A_n\}$, n = 0,1,.. be the time-instants when the state of the computation changes. Define a vector sequence $Y = \{Y_n\}$, n = 0,1,.. where $Y_n = (I, w_I)$ = (set of nodes ready and waiting at transition time $A_n^-$, the delay incurred by these nodes thus far), where $A_n^-$ denotes the time just prior to the $n$th transition time. Define the sequence $Z = \{Z_n\}$, n = 0,1,.. where $Z_n = (J, r_J)$ = (set of nodes running at $A_n^-$, remaining execution time of these nodes). Sequences $Z$ and $Y$ capture information about running and waiting nodes. Define a sequence $U = \{U_n\}$ n = 0,1,.. where $U_n = \{(App, m^{App}, t^{App})$ for all applications$\}$ = (application, adaptation level of application at $A_n^-$, elapsed time of application at $A_n^-$). $t^{App}$ is set to zero at the start of each application iteration, and is incremented at transitions when nodes in that iteration finish running or finish waiting. ($Y$, $Z$, $U$) models the state of computation in the system and is called the computation process.

Figure 5 illustrates the computation process for a simple example with two concurrent applications. For simplicity of illustration, suppose that the distribution of execution time for node 4 is $\{(t_{4a}, pa), (t_{4b}, 1\text{-}pa)\}$. All other nodes $i$ are assumed to have deterministic execution times $t_i$. Initially, nodes 1 and 2 start running on processors $P1$ and $P3$. The state at $A_0$ is: $\{(), ((1,t_1), (2,t_2)), (1, m_1, 0), (2,$

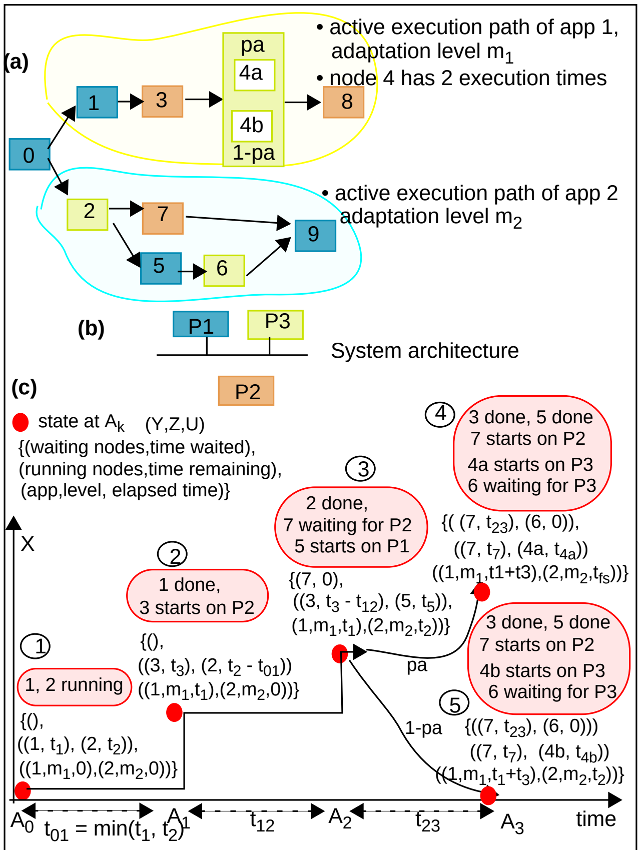
(a)
• active execution path of app 1, adaptation level $m_1$
• node 4 has 2 execution times
0
1
3
pa
4a
4b
1-pa
8
• active execution path of app 2 adaptation level $m_2$
2
7
9
5
6
(b)
P1
P3
System architecture
P2
(c)
● state at $A_k$ (Y,Z,U)
{(waiting nodes,time waited),
(running nodes,time remaining),
(app,level, elapsed time)}
X
1
1, 2 running
{(), ((1, $t_1$), (2, $t_2$)), ((1,$m_1$,0),(2,$m_2$,0))}
2
1 done, 3 starts on P2
{(), ((3, $t_3$), (2, $t_2$ - $t_{01}$)) ((1,$m_1$,$t_1$),(2,$m_2$,0))}
3
2 done, 7 waiting for P2 5 starts on P1
{(7, 0), ((3, $t_3$ - $t_{12}$), (5, $t_5$)), (1,$m_1$,$t_1$),(2,$m_2$,$t_2$))}
4
3 done, 5 done 7 starts on P2 4a starts on P3 6 waiting for P3
{( (7, $t_{23}$), (6, 0)), ((7, $t_7$), (4a, $t_{4a}$)) ((1,$m_1$,t1+t3),(2,$m_2$,$t_{fs}$))}
pa
1-pa
5
3 done, 5 done 7 starts on P2 4b starts on P3 6 waiting for P3
{((7, $t_{23}$), (6, 0))) ((7, $t_7$), (4b, $t_{4b}$)) ((1,$m_1$,$t_1$+$t_3$),(2,$m_2$,$t_2$))}
$A_0$
$t_{01}$ = min($t_1$, $t_2$)
$A_1$
$t_{12}$
$A_2$
$t_{23}$
$A_3$
time

**Figure 5. a) Selected execution paths for two apps b) architecture, mapping c) Computation process.**

$m_2$, 0)} which is elaborated as: {(no waiting nodes), (1 running with remaining time $t_1$, 2 running with remaining time $t_2$), (application 1 in level $m_1$ and elapsed time 0, application 2 in level $m_2$ and elapsed time 0). Suppose that $t_1 < t_2$. At $t_1$, 1 finishes and there is a state change in the system. Only one transition is possible at this time: 3 starts running on *P2* with remaining time $t_3$. 2 is still running with remaining time $t_2$-$t_1$. At time $t_2$, 2 finishes. 5 and 7 are "ready". Since 7 is mapped to *P2* which is currently running 3 (tasks are non-preemptive), 7 goes on the waiting list, while 5 starts running. Suppose that 3 and 5 finish at $t_1$+$t_3$, 7, 6, and 4 are ready to run. Since 4 and 6 are mapped to the same processor, the run-time scheduler selects one based on priorities. Suppose that a static priority scheme is used, where application 1 has a higher priority. In this case, node 4 is set to run and node 6 goes on the waiting list. (Other scheduling policies can similarly be modeled. Note that the selection of the next node to run is typically made using information carried in the current state only; thus the computation process retains the Markovian property.) There are two possible execution times for 4, leading to two possible states at this point, with probabilities *pa* and 1-*pa*. This process is continued until all the applications end at the same time instant. This is called the regeneration point, after which the computation process repeats its evolution. Note that the dependencies in execution times across different nodes can be modeled in this framework by introducing additional paths. Although this increases the size of the state space, the method is quite fast and has been used to analyze realistic systems quite efficiently, as demonstrated in the results section.

We now state the following theorem that characterizes the computation process. We omit the proof due to lack of space.

**Theorem**: *The joint sequence (Y, Z, U) represents the state of computation at each transition and is a Markov chain. By joining the state of computation between transitions by straight lines, we obtain the computation process X. X is a continuous-time stochastic process and since its underlying jump process (Y, Z, U) is Markovian, X is a semi-Markov process* [18][19].

The implications of this theorem are that the performance statistics can be computed on this process by analyzing the behavior of a single traversal through the state space of X using the Markov renewal theory. This is much faster than simulating the system, since several simulation runs are needed to get reliable statistics of the processing delay, due to variable execution times. Our approach is much faster since a single exploration of the state space can be done efficiently even for a large state space. Further, this approach is much more efficient than the convolution approach [8][9], which is especially unwieldy for arbitrary distributions.

Also, for a given set of task-level distributions, the processing delay distribution computed by the method is exact. We are currently evaluating how effectively these distributions capture real applications. The simulations done so far show a close match with computed results.

A second key observation is that the computation process is Markovian under several scheduling policies. An exact proof is omitted for lack of space. Consequently, the impact of different scheduling policies can be analyzed within this framework.

## 3.4. Performance metrics

We first compute the stationary distribution of the computation process. Roughly speaking, the stationary distribution specifies the probability of a transition into each state, given that a jump occurs.

**Stationary distribution of (*Y, Z, U*)** We have stated earlier that, for a particular adaptation value *M*, the joint process (*Y, Z, U*) is a discrete-time Markov chain. This means that when (*Y, Z, U*) jumps from state (*i, j, a*) to (*k, l, c*) its future evolution is independent of the past, given state (*i, j, a*). This chain is therefore completely specified in terms of its transition probability function defined by $R_{ija}^{klc}$, which is the probability that (*Y, Z, U*) moves from a state (*i, j, a*) to (*k, l, c*) in a single jump. This one-step probability can be determined for a particular set of applications from the computation process. The stationary distribution $\pi_{ija}^{M}$ is the probability that (*Y, Z, U*) jumps to state (*i, j, a*), given that (*Y, Z, U*) changes states and can be computed from *R*, since it satisfies the equations: $\Sigma_{(i,j,a) \text{ in } S}\, \pi_{ija}^{M} R_{ija}^{klc} = \pi_{klc}^{M}$ for all (*k,l,c*) in state space *S*, and $\Sigma_{(k,l,c) \text{ in } S}\, \pi_{klc}^{M} = 1$.

To obtain the stationary distribution unconditioned of adaptation level *M* we assume that the adaptation process converges to a steady-state distribution γ, as described earlier. Here, $\gamma_M$ is roughly the long-term fraction of time the adaptation process spends in level *M*. γ can be computed from the transition matrix of the adaptation process and the interval of RTCP reports. Assuming γ is determined, the unconditioned stationary distribution $\pi_{ija} = \Sigma_{all\ M}\, \pi_{ija}^{M} \gamma_M$. Next we derive an expression for the processing delay distribution, which gives the probabilities for different values of processing delay for an application. This is a powerful result.

**Processing Delay distribution** : Suppose we wish to determine the probability that the processing delay (PD) of a particular execution path *m* of an application exceeds a value *T*. Let node *b* be the predecessor to the spacer node on the execution path *m*. Then, Pr{PD of path m > *T*} = *max*[ Pr{PD of path m > *T*, for all M}]. Note that the right hand side is a *max* operator over all adaptation vectors of which path *m* is an element. The expression inside the *max* operator is the ratio of the number of times the processing delay of the path *m* exceeds *T* to the number of times path *m* is activated. Recall that $\pi_{ija}{}^{M}$ represents the probability of a jump of the underlying process (*Y*, *Z*, *U*) to the state (*i*, *j*, *a*) conditioned on a jump. If we focus only on the jumps to state where node *b* begins running, it is clear that:

$$\Pr\{\text{PD of path m} > \text{T}\} = \frac{\text{number of times PD} > \text{T}}{\text{number of times path m activated}}$$

$$= \frac{\sum_{\forall ija} \pi_{ija}{}^{M} \Big|_{b \in J,\, r_b = t_b,\, t^{App} + r_b > T}}{\sum_{\forall ija} \pi_{ija}{}^{M} \Big|_{b \in J,\, r_b = t_b}} .$$

Having computed the processing delay of path *m* in the application, the processing delay of the entire application is computed by choosing the worst-case processing delay among all paths.

**Excess delay probability** $Pr_{excess}$: The probability of processing delay exceeding the deadline *D*, is simply computed by setting *T* equal to *D* in the above expression.

The above analytical framework has been implemented as a software prototype called *AsaP*. *AsaP* is about 10k lines of C code and has a Tcl/Tk GUI. Applications can be specified either textually or by using the Ptolemy [21] front-end. The state space for a given set of applications is generated using a recursive procedure. The transition probability matrix *R* is computed from the generated state space. The Jacobi iterative power series method [20] is used to solve for the stationary distribution π. Several other performance metrics such as nodal wait and resource utilization can also be derived.

# 4. Results

We illustrate the use of *AsaP* for system-level design with three examples. In the first example, the processing delay for an adaptive video application is computed for different adaptation levels. This is used to compute the maximum sustainable adaptation level. In the second example, the variation in the processing delay due to multiple concurrent applications is quantified. This shows how to trade off quality and computation between audio and video applications. In the third example, the impact of different run-time scheduling policies on the processing delay is studied. In all three cases, we demonstrate results for a single processor system, although the analytical framework can also handle multiprocessor systems.

## 4.1. Example 1: Impact of adaptation levels on processing delay of MPEG

We use a MPEG video encoder as an example of an adaptive application, where adaptation levels correspond to frame rates of 7.5fps, 10 fps, and 15 fps. Figure 6 shows the application and the execution time distributions. The execution times for each node were obtained by profiling the Berkeley MPEG-1 video encoder [22] on a table-tennis sequence. We are currently exploring systematic methods to compute task-level distributions.

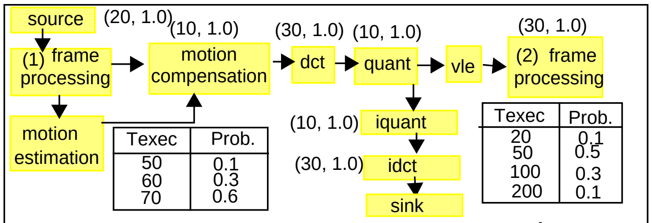

**Figure 6. MPEG video encode. Exec. time dist. in $10^4$ cycles.**

The adaptation level changes were profiled based on experimental results reported in [3], where a video conferencing application VIC [23] is modified to adapt its bit rate based on network feedback. The steady state distribution for each adaptation level was calculated by profiling the adaptation level changes over a period of 300 seconds (Table 1). Although such a profile should actually be computed over a much larger time window, this is an illustrative example.

The output of *AsaP* is shown in the last two columns of Table 1. For example, in level 3, the excess delay probability is 0.1, i.e. 10% of the samples miss their deadlines. The last column in Table 1 represents the effective miss probability for each level, by factoring in the probability of the level itself. While the deadlines are missed quite often in level 4, level 4 itself is much less frequent. Figure 7 plots the probability of deadline misses for different adaptation levels on two architectures, *arch1* and *arch2*. The processing delay indicates sustainable adaptation levels. Thus, for a 10% acceptable miss rate, *arch1* supports adaptation levels 1 through 3, while *arch2* supports levels 1 and 2 only.

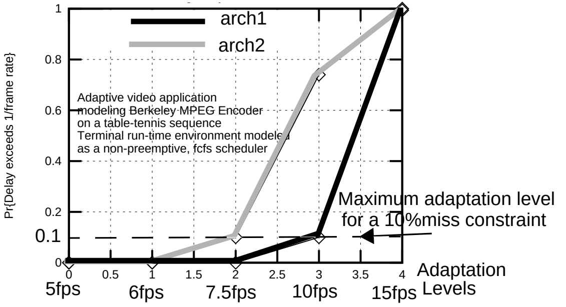

**Figure 7. Excess delay probability vs. adaptation levels for two architectures. *arch1* 1.3 times as fast as *arch2*.**

| Adaptation Level | steady state distribution γ | $Pr_{excess}$ | γ x $Pr_{excess}$ |
|---|---|---|---|
| 0 (5 fps) | 0.0727 | 0.0 | 0.0 |
| 1 (6 fps) | 0.0909 | 0.0 | 0.0 |
| 2 (7.5 fps) | 0.2181 | 0.0 | 0.0 |
| 3 (10 fps) | 0.5454 | 0.1 | 0.05454 |
| 4 (15 fps) | 0.0727 | 0.994 | 0.0722 |

**Table 1: (Ex. 1) Adaptation levels (frame rates), steady-state distribution of adaptation process, excess delay probability.**

## 4.2. Example 2: Impact of adaptation levels on processing delay of concurrent applications

In this experiment, we consider two concurrent applications: video (adapting between 7.5 fps and 15 fps) and audio (adapting between PCM and LPC, with a constant period of 20ms). Table 2 summarizes the processing delay for different combinations of audio algorithms and video frame rates. This data is used to determine feasible combinations of adaptation levels. Consider the combination (LPC, 7.5fps), with audio missing 24% of its deadlines. If a higher frame rate is desired, switching to (LPC, 15fps) leads to an unacceptable 46% miss rate for audio. The application manager should instead step down to (PCM, 15fps). Such statically computed information may be used to design “smart” application managers.

| (audio, video) | (PCM, 7.5fps) | (PCM,15fps) | (LPC,7.5fps) | (LPC,15fps) |
|---|---|---|---|---|
| (Deadlines $10^4$ cycles) | (200, 1330) | (200, 660) | (200, 1330) | (200, 660) |
| $\mathrm{Pr}_{excess}$ | (0.00046, 0) | (0.01, 0) | (0.241, 0) | (0.462, 0.007) |
| $\mathrm{PD}_{ave}$ in $10^4$ cycles | (16.2, 344.1 | (32.1, 344.1) | (146.2, 509.7) | (198, 500.93) |
| $\mathrm{PD}_{max}$ in $10^4$ cycles | (240, 486) | (240, 486) | (440, 675) | (450, 680) |
| Number of states | 19856 | 16070 | 6457 | 20767 |
| State space time | 1393.29 s | 856.96 s | 122.16 s | 1446.27 s |
| Analysis time | 1948.53 s | 862.37 s | 494.03 s | 352.11 s |

**Table 2: (Example 2) Performance metrics for audio (PCM, LPC) and video (7.5fps, 15fps).**

## 4.3. Example 3: Impact of run-time schedulers

Table 3 summarizes the processing delay distribution when video is at 15fps and audio uses LPC (LPC 15fps) under three different run-time scheduling policies. In FCFS, nodes are set to run in the order in which they become “ready”. RMS and EDF are prioritized schemes, where nodes with higher priority are selected to run first. In RMS, priorities of the nodes are set according to rates. In this particular example, audio has a higher priority over video. In EDF, the priorities are computed at run-time; the node whose “due date” is earliest gets higher priority. Note how shifting from FCFS to RMS improves the mean and worst-case delays and probability of deadline misses for audio, while degrading the performance of video. Using EDF seems to be a good compromise for both applications. To reiterate the importance of a distribution of the processing delay against just the worst case value, consider the (LPC, 15fps) case for RMS scheduler in Table 2. The worst case processing delay is 290. However, the distribution tells us that the mean is 118, the standard deviation is 29, and only 2% of all samples miss deadlines. This is much more valuable than knowing that the worst case execution time is 290. Typically, CM applications have constraints such as “no more than 10% samples should miss deadlines”. Using worst-case leads to very conservative and hence inefficient architecture design.

| Perf. metrics (audio, video) | FCFS | RMS | EDF |
|---|---|---|---|
| $\mathrm{Pr}_{excess}$ | (0.462, 0.007) | (0.0235, 0.52) | (0.175, 0.227) |
| $\mathrm{PD}_{ave}$ | (198.0, 501.0) | (118.4, 685.9) | (169.3, 615.7) |
| $\sigma$ | (77.5, 54.3) | (29.34, 94.72) | (61.59, 58.07) |
| $\mathrm{PD}_{max}$ | (450,680) | (290,980) | (380,840) |

**Table 3: Example 3: Impact of run-time scheduling policy on the performance delay of LPC audio and video at 15fps**

## 5. Conclusions

We have identified system-level design issues for next-generation networked embedded systems in the context of adaptive applications and run-time schedulers. Applications adapt in order to reduce the output bit rate. However, adaptations impose different processing demands on the end terminal. This variation, in addition to task-level processing variations, manifests by making the processing delay of each application variable and unpredictable. Run-time scheduling policies also impact the performance of the end-system. We propose an analytical framework to estimate the performance of a set of concurrent adaptive applications for a particular architecture and run-time scheduler. The framework can efficiently compute the exact distribution of the processing delay of applications. This result can be used to synthesize cost-efficient solutions for systems requiring soft real-time guarantees. We demonstrate the use of this framework to explore the system-level tradeoffs in selecting adaptation levels for applications, designing smart application managers, and in selecting an appropriate run-time environment.

## 6. References

[1] PDA Buyer’s Guide, Pen Computing Magazine, vol. 3, no. 11, July/August 1996, pp. 84.

[2] H. Schulzrinne, *et al*. “RTP: A Transport Protocol for Real-Time Applications”, RFC 1889, Audio/Video Transport WG, Jan. 1996.

[3] I. Busse, *et al*, “Dynamic QoS Control of Multimedia Applications based on RTP”, *Computer Communications* 19:1, Jan. 1996, pp. 49-58.

[4] J.-C. Bolot, A. Vega-Garcia, “Control Mechanisms for Packet Audio in the Internet”, *Proc. IEEE Infocom ‘96*, CA, April 1996, pp. 232-9.

[5] Y. S. Li, S. Malik, A. Wolfe, “Performance Estimation of Embedded Software with Instruction Cache Modeling”, *ICCAD 95*.

[6] R. K. Gupta, *Co-synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic, 1995.

[7] J. Henkel *et al*, “The Interplay of Run-time Estimation and Granularity in HW/SW Partitioning”, *Proc. Codes96*, USA, March 1996, pp. 52-8.

[8] J. Kim, K. G. Shin, “Execution Time Analysis of Communicating Tasks in Distributed Systems”, *IEEE Trans. on Computers*, vol. 45, no. 5, May 1996, pp. 572-9.

[9] Y. Li, J. Antonio, “Estimating the Execution Time Distribution for a Task Graph in a Heterogeneous Computing System”, *Proc. Sixth Heterogeneous Computing Workshop* (HCW ‘97), Switzerland, pp. 172-84.

[10] Ti Yen, W. Wolf, “Performance Estimation for Real-time Distributed Embedded Systems”, *Proc. of ICCD 95*, pp. 64-69.

[11] C. L. Liu, J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment”, *JACM*, v20, 1973, pp. 46-61.

[12] K. Ramamritham *et al*., “Scheduling Algorithms and Operating Systems Support for Real-Time Systems”, *Proc. of the IEEE*, vol. 82, no. 1, Jan. 1994, pp. 55-66.

[13] D. K.Y. Yau, *et al*. “Adaptive Rate-Controlled Scheduling for Multimedia Applications”, *Proc. ACM Multimedia’96*, Boston, MA, Nov. 1996.

[14] R. Yavatkar, K. Lakshman, “A CPU Scheduling Algorithm for Continuous Media Applications”, *Proc. of NOSSDAV*, April 1995, pp.223-6.

[15] V. Mooney *et al*., “Run-Time Scheduler Synthesis for Hardware-Software Systems and Application to Robot Control Design”, *CODES 97*.

[16] F. Balarin *et al*., “Automatic Generation of a Real-Time Operating System for Embedded Systems”, *Proc. of CODES 97*.

[17] P. Chou *et al*., “Software Scheduling in the Co-synthesis of Reactive Real-time Systems”, *Proc. of DAC*, June 1994, pp 1-4.

[18] E. Cinlar, *Introduction to Stochastic Processes*, Prentice-Hall, 1975.

[19] S. Ross, *Introduction to Probability Models*, Academic Press., 1985.

[20] V. L. Wallace, R. S. Rosenberg, “Markovian Models and Numerical Analysis of Computer System Behavior”, AFIPS Spring Joint Computer Conf. Proceedings, pages 141-148, 1966.

[21] Ptolemy at *http://ptolemy.eecs.berkeley.edu*

[22] *http://bmrc.berkeley.edu/projects/mpeg/mpeg_encode.html*

[23] S. McCanne, V. Jacobson, “*vic*: a flexible framework for packet video”, *Proc. of ACM Multimedia* ‘95, CA, Nov 1995, pp. 511-522.