Using Neural Networks for Quality Management

Jacques Combaz Jean-Claude Fernandez Mohamad Jaber
Loïc Strus
Verimag, Centre Equation - 2 avenue de Vignate F38610 Gières, France

Abstract

We present a method for fine grain QoS control of multimedia applications. This method takes as input an application software composed of actions. The execution times are unknown increasing functions of quality level parameters. Our method allows the construction of a Quality Manager which computes adequate action quality levels, so as to meet QoS requirements for a given platform. These include requirements for safety (action deadlines are met) as well as optimality (maximization and smoothness of quality levels).

In this paper, we use learning techniques for computation of quality management policies. Given input parameters of the actions, a neural network is used to refine online pre-computed average execution times. Using refine average execution times allows a better control of the application, which leads to a reduction of fluctuations of CPU load.

We present experimental results including the implementation of the method and benchmarks for an MPEG4 video encoder.

1. Introduction

Designing systems meeting both hard and soft real-time requirements is a challenging problem. There exist well-established design methodologies for hard real-time systems, that is systems that do not violate critical properties such as deadlines. These methodologies rely on complete understanding and formal analysis of the behavior of the system. As a consequence, they are based on worst-case analysis using conservative approximations of the system dynamics and static resource reservation. This implies high predictability but a non optimal use of resources.

In contrast, design methodologies for soft real-time are based on average-case analysis and seek more efficient use of resources (e.g. optimization of speed, jitter, memory, bandwidth, power) without addressing critical behavior issues. They are used for applications where some degradation or even temporal denial of service is tolerated, e.g., multimedia and telecommunications.

These two classes of design methodologies are currently disjoint. Meeting hard real-time properties and making optimal use of available resources seem to be two antagonistic requirements. The existing gap between hard and soft real-time often leads to costly and unreliable solutions.

Adaptivity is a means to bridge the gap between the two classes of design methodologies. Fine grain QoS control can be used to increase predictability of execution times, and thus modify the worst-case behavior of the system. It allows drastic reduction of the impact of worst-case execution times on resource utilization [3, 4, 5]. Nevertheless, non flexibility of approaches based on worst-case execution times may still lead to non-optimal use of available resources. Development of soft real-time approaches that ensure predictability is a key challenge in the design of modern methologies for real-time embedded systems.

Our method targets multimedia applications. It allows adapting the overall system behavior by adequately setting quality level parameters for its actions. The objective of the quality management policy is to meet QoS requirements including three types of properties: 1) safety (no deadlines are missed); 2) optimality, (maximization of the utilization of available time budget); 3) smoothness of quality levels.

The method takes as input an application software with timing information about its actions. This includes deadlines and (platform-dependent) execution times. It produces a controlled application software meeting the QoS requirements for the target platform. This is obtained by applying to the application software a *Controller* consisting of a *Scheduler* and a *Quality Manager*. Depending on the progress of the computation, the Scheduler chooses the next action to be executed and the Quality Manager the associated quality level parameter.

In [4], we explained how to build quality management policies meeting QoS requirements. These are computed from average and worst-case estimates of execution times. We also provided low overhead implementations of the controller in [5].

In this paper, we use learning techniques for computation of quality management policies. Given input parameters of the actions, a neural network is used to refine online pre-computed average execution times. Using refine average execution times allow a better control of the

application, which leads to a reduction of fluctuations of CPU load.

We consider the following simplified version of the general problem by assuming that the application software is already scheduled (see Figure 1):

• The application software cyclically performs input/output transformations of data streams. It is described as a finite sequence of actions (C-functions). Its execution during a cycle can be controlled by choosing *quality level parameters*. We assume that the execution times of actions are unknown and are increasing with quality.

• We consider single-thread implementations of the application software on a platform for which it is possible, by using timing analysis and/or profiling techniques, to compute average estimates of execution times of actions for different quality levels. Action execution is assumed to be atomic. A compiler is used to generate the controlled software from the initial application software, for given deadline requirements and execution times.

The controlled software can be considered as the composition of the initial application software with a *Quality Manager* (see Figure 2). The latter monitors the progress of the computation within a cycle of the application software. At any state of the cycle, it chooses the quality level for the next action to be executed, guided by a *quality management policy*.

This is a constraint guaranteeing safety and embodying an optimality criterion. The Quality Manager chooses the maximal quality satisfying this constraint.

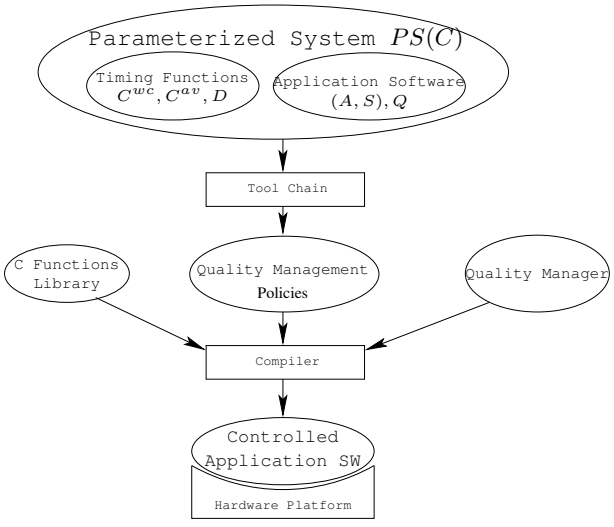

Figure 1. Prototype tool implementation

Our method significantly differs from existing ones. The main difference is fine granularity of quality management, which allows a better controllability of the application. Most existing techniques are applied at system or task level, focus on average scenario and do not provide predictability. Buttazzo et al.'s elastic tasks model [2], as well as slack scheduling [6], [11] and gain time techniques [1] are based only on worst-case execution times and do not deal with quality smoothness. Buttazzo et al. propose the elastic tasks model [2], but their approach is based on worst-case execution times. Slack scheduling and gain time techniques [6], [11] can be used to determine the quality level of the tasks of a system, but it is also based only on worst-case execution times and thus cannot deal with quality smoothness property. A common and simple way to treat CPU overload is to skip an instance of a task [9]. Lu et al. [13] propose a feedback scheduling based on PID controllers. Steffens et al. [18], [14] minimize deadline misses of an MPEG decoder by applying a Markov decision process and reinforcement learning techniques, combined with structural load analysis.

The paper is organized as follows. In section 2 we present the quality management problem. Neural networks are presented in section 3. Section 4 presents experimental results for a non trivial MPEG4 video encoder.

2. Quality Management

2.1. Definition of the Problem

We provide a formalization of the quality management problem by considering that the application software is already scheduled. It is characterized by an execution sequence $\{ s_{i-1} \xrightarrow{a_i} s_i \}_{1 \leq i \leq n}$, where $S = \{ s_0, \ldots, s_n \}$ is a set of *states* and for all i we have $a_i \in A$ where A is a finite set of *actions*. Actions correspond to blocks of code. Their execution is atomic.

The execution of the application software (A, S) on a platform, is modeled by an *execution time function* $C : A \to \mathbb{R}^+$, associating with an action a_i its execution time $C(a_i)$. The corresponding timed execution sequence is $\{ (s_{i-1}, t_{i-1}) \xrightarrow{a_i} (s_i, t_i) \}_{1 \leq i \leq n}$ such that $t_0 = 0$ and $t_i - t_{i-1} = C(a_i)$.

Execution times for actions may considerably vary over time as they depend on the contents of data. Furthermore, non predictability of the underlying platform is an additional factor of uncertainty. We consider that they are not known in advance, but are bounded by worst-case estimates. To cope with the inherent uncertainty of execution times, we assume that some actions – called *controllable* – are parameterized by quality levels. This leads to the following model.

Definition 1 *A* **parameterized system** PS *is an application software* (A, S) *with*

• *a finite set of integer* quality levels Q

• *the set of actions* A *is partitioned into* $A^{\mathbf{c}}$ *and* $A^{\mathbf{u}}$*, where* $A^{\mathbf{c}}$ *is the subset of* controllable *actions, and* $A^{\mathbf{u}}$ *is the subset of* uncontrollable *actions*

• *a* worst-case *execution time function* $C^{wc} : A \times Q \to \mathbb{R}^+$ *non-decreasing with quality levels for controllable actions and constant with quality levels for uncontrollable actions, that is, for all actions* $a_i \in A^{\mathbf{c}}$*, the function* $q \mapsto C^{wc}(a_i, q)$ *is a non-decreasing function, and for all actions* $a_i \in A^{\mathbf{u}}$ *the function* $q \mapsto C^{wc}(a_i, q)$ *is constant*

• *a parameter* C*, called* actual *execution time function,*

$C : A \times Q \rightarrow \mathbb{R}^+$ *non-decreasing with quality levels for controllable actions and constant with quality levels for uncontrollable actions, and such that* $C(a_i, q) \leq C^{wc}(a_i, q)$ *for any action* a *and quality level* q.

The execution of a parameterized system is characterized by the family of sequences $\{ (s_{i-1}, t_{i-1}) \xrightarrow{a_i, q_i} (s_i, t_i) \}_{1 \leq i \leq n, q_i \in Q}$ *such that* $t_0 = 0$ *and* $t_i - t_{i-1} = C(a_i, q_i)$.

Quality Managers are used to adequately restrict the behavior of a parameterized system so as to meet given properties.

Definition 2 *Given a parameterized system* PS *a* **Quality Manager** *is a function* $\Gamma : S \times \mathbb{R}^+ \rightarrow Q$ *giving, for a state* (s_{i-1}, t_{i-1}) *of* PS*, the quality level* q_i *for executing the next action* a_i.

$PS||\Gamma$ *denotes a* controlled system *obtained as the composition of the parameterized system* PS *and the Quality Manager* Γ. *For a given actual execution time function* C*, it has a single execution sequence* $\{ (s_{i-1}, t_{i-1}) \xrightarrow{a_i, q_i} (s_i, t_i) \}_{1 \leq i \leq n}$ *such that* $q_i = \Gamma(s_{i-1}, t_{i-1})$.

The quality management problem for a given parameterized system PS consists in finding a Quality Manager Γ meeting the QoS requirements. That is,there are no deadline misses and the overall quality is maximal. It is formalized as follows.

Definition 3 (quality management problem) *Given a parameterized system* PS *and a* deadline function $D : A \rightarrow \mathbb{R}^+$ *associating with each action* a_i *its deadline* $D(a_i)$*, find a Quality Manager* Γ *such that for any actual time function* $C \leq C^{wc}$*:*

• Γ *is* safe *(deadlines are met), that is for any state* (s_i, t_i) *of* $PS||\Gamma$ *we have* $D(a_i) \geq t_i$ *where* $t_i = C(a_1, q_1) + \ldots + C(a_i, q_i)$.

• *The overall execution time is maximal, that is for any safe Quality Manager* Γ'*,* $t_n \geq t'_n$*, where* t_n *(resp.* t'_n*) is the completion time of the last action in* $PS||\Gamma$ *(resp.* $PS||\Gamma'$*).*

In [4], we require in addition to feasibility and optimality, *smoothness* for the quality levels chosen by the Quality Manager. Informally, smoothness means low fluctuation of quality levels. Due to lack of space, we do not study this property which is essential for most multimedia applications [16].

2.2. Quality Manager Design

We summarize the method for the design of Quality Managers, given in [4].

Quality Manager Design Principles

Figure 2 shows interaction between the Quality Manager Γ, applying a *quality management policy*, and the application software, i.e. the parameterized system PS. The

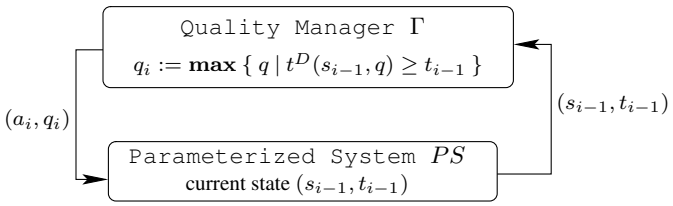

Figure 2. Quality Manager

Quality Manager observes the current state (s_{i-1}, t_{i-1}) of PS and computes the next quality level q_i for the next action a_i. The Quality Manager is defined by:

$$\Gamma(s_{i-1}, t_{i-1}) = q_i = \mathbf{max}\ \{\ q \mid t^D(s_{i-1}, q) \geq t_{i-1}\ \}.$$

The function $t^D : A \times Q \rightarrow \mathbb{R}^+$ defines the *quality management policy* of the Quality Manager. It gives for a state of the application software s_{i-1} and a quality level q, the estimated elapsed time $t^D(s_{i-1}, q)$ at next state s_i if the rest of the actions is executed with constant quality q. If the inequality $t^D(s_{i-1}, q) \geq t_{i-1}$ is satisfied, then it is possible to complete execution without missing the deadlines specified by D. The chosen quality level q_i at state (s_{i-1}, t_{i-1}) is maximal amongst the quality levels q meeting the inequality $t^D(s_{i-1}, q) \geq t_{i-1}$.

The function t^D is defined as follows:

$$t^D(s_{i-1}, q) = \mathbf{min}_{i \leq k \leq n}\ D(a_k) - C^D(a_i..a_k, q),$$

where $C^D(a_i..a_k, q)$ denotes an estimation of the total execution time for the sequence of actions a_i, a_{i+1}, ..., a_k.

Choosing an adequate quality management policy, i.e., that ensures safety and an optimal use of resources, is a non trivial problem discussed in [3] and [4]. In [4] we proposed the *mixed* quality management policy based on the *mixed* execution time function C^{mx} defined below. Its interest has been shown through both theoretical and experimental results.

Quality Management Policies

We use execution time function C^D that combines two execution time functions C^{sf} and C^{av}. The first allows respecting the safety requirement, that is no deadline is missed. The second is used to enhance smoothness of quality levels.

The *safe* execution time function C^{sf} gives a worst-case estimation of the total execution time of $a_i..a_k$:

$$C^{sf}(a_i..a_k, q) = C^{wc}(a_i, q) + C^{wc}(a_{i+1}..a_k, q_{min})$$

where $C^{wc}(a_{i+1}..a_k, q_{min})$ denotes the total worst-case execution time for the sequence of actions a_{i+1}, ..., a_k with the minimal quality level $q_{min} = \mathbf{min}\ Q$. That is,

$$C^{wc}(a_{i+1}..a_k, q_{min}) = \sum_{i+1 \leq j \leq k} C^{wc}(a_j, q_{min}).$$

As the quality level can be changed by the Quality Manager after the execution of the first action a_i, we take the quality level q for the first action, and q_{min} for the remaining actions. The application of *safe* quality management policy ensures safety of the Quality Manager. Nevertheless, it may lead to considerable variations of quality levels in a sequence e.g., by starting with high quality levels and terminating with low quality levels.

To improve smoothness of the quality levels, we introduce an *average* execution time function $C^{av} : A \times Q \rightarrow \mathbb{R}^+$, non-decreasing with quality. Average execution times can be estimated by static analysis and/or profiling techniques. We define δ^{max} as the maximum difference between the worst-case and the average behavior:

$$\delta^{max}(a_i..a_k, q) = \mathbf{max}_{i \leq j \leq k}\ \delta(a_j..a_k, q),$$

where $\delta(a_j..a_k, q) = C^{sf}(a_j..a_k, q) - C^{av}(a_j..a_k, q)$. That is, for a sequence of actions $a_i..a_k$ and quality level q, $\delta^{max}(a_i..a_k, q)$ is a kind of safety margin with respect to the average behavior. It measures uncertainty on execution times in order to meet the deadlines.

The mixed execution time function C^{mx} is defined by $C^{mx} = C^{av} + \delta^{max}$. It combines average and worst-case behavior. It is possible to take into account execution time needed for quality management by adequately overestimate average and worst-case execution times.

2.3. Impact of Worst-Case Execution Times

Using worst-case execution times may lead to an overestimation of the time budget needed to execute the application software at a given quality level. The function δ^{max} defined above give an estimation of the possible loss of time budget when using mixed quality management policy. In the following, we introduce functions — η and β — computed from average and worst-case execution times.

Definition 4 *We define the functions* $\eta : A \times Q \rightarrow \mathbb{R}^+$ *and* $\beta : A \times Q \rightarrow \mathbb{R}$ *as follows:*

$$\begin{aligned} \eta(a_i, q) &= C^{wc}(a_i, q) - C^{av}(a_i, q) \\ \beta(a_i, q) &= C^{wc}(a_i, q_{min}) - C^{av}(a_i, q). \end{aligned}$$

For an action a_i and a quality level q, $\eta(a_i, q)$ is the difference between the worst-case and the average execution time. The value η can be considered as the uncertainty for the execution time of the action a_i for quality q. The value $\beta(a_i, q)$ is the difference between the worst-case execution time for the action a_i at the minimal quality level q_{min}, and the average execution time for the actions at the quality level q. It is related to the "fall-back" capability of a_i for quality q: for small values of β, in particular negative values, the controller can speed up the application by selecting the minimal quality level, even if we consider the worst-case assumption (i.e. $C = C^{wc}$). Then, we write δ as follows:

$$\delta(a_1..a_n, q) = \eta(a_1, q) + \beta(a_2..a_n, q)$$

where $\beta(a_2..a_n, q) = \beta(a_2, q) + \ldots + \beta(a_n, q)$.

Consider a set of quality levels $Q = \{\ 0, 1\ \}$ and a set of actions $A = \{\ a, b\ \}$ such that worts-case and average execution time functions are given in Figure 3. Notice that a is a controllable action and b is an uncontrollable action, that is, $A^{\mathbf{c}} = \{\ a\ \}$ and $A^{\mathbf{u}} = \{\ b\ \}$.

$C^{wc}(x,q)$	$x=a$	$x=b$
$q=0$	4	6
$q=1$	10	6

$C^{av}(x,q)$	$x=a$	$x=b$
$q=0$	2	3
$q=1$	7	3

$\eta(x,q)$	$x=a$	$x=b$
$q=0$	2	3
$q=1$	3	3

$\beta(x,q)$	$x=a$	$x=b$
$q=0$	2	3
$q=1$	-3	3

Figure 3. Worst-case and average execution time functions.

Consider an application software composed of p actions a and p actions b. Assume that there exists two possible schedules for this application — schedule #1 and schedule #2 (see Figure 4) — corresponding to the following execution sequences:

schedule #1: $s_0 \xrightarrow{a} s_1 \xrightarrow{b} \ldots \xrightarrow{a} s_{2p-1} \xrightarrow{b} s_{2p}$
schedule #2: $s_0 \xrightarrow{a} \ldots \xrightarrow{a} s_p \xrightarrow{b} \ldots \xrightarrow{b} s_{2p}$.

It can easily be shown that:

$$\delta^{max}(\underbrace{ab\ldots ab}_{p \text{ times}}, 1) = \eta(a,1) + \beta(b,1) = 6 \text{ and}$$

$$\delta^{max}(\underbrace{a\ldots a}_{p \text{ times}}\ \underbrace{b\ldots b}_{p \text{ times}}, 1) = \eta(a,1) + p\beta(b,1) = 3(p+1).$$

This means that δ^{max} for quality $q = 1$ and schedule #1 is constant, whereas it tends to $+\infty$ as p increases for schedule #2.

The difference between the values of δ^{max} obtained with schedules #1 and #2 comes from the position of controllable actions in the schedule. Controllable actions are scattered all along schedule #1, whereas they are put together at the beginning of schedule #2 (see Figure 4). Consequently, the Quality Manager keeps control on the execution times of actions during the execution of schedule #1. On the contrary, once all controllable actions —

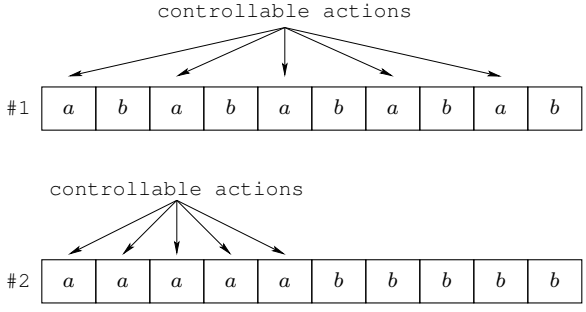

Figure 4. Schedules #1 and #2 for $p = 10$

actions a — have been executed in schedule #2, the Quality Manager has no control anymore on the (uncertain) execution times of the remaining (uncontrollable) actions. Uncontrollability combined with unpredictability leads to an overestimation of the execution time of the remaining actions — actions b —, that is, considering their worst-case estimates.

For such a schedule considering worst-case scenario lead to non-optimal use of available resources, and make mixed quality management policy unapplicable without loss of time budget. Using only (fixed) average execution times, that is, the quality management policy $C^D = C^{av}$, may lead to fluctuations of CPU load and frame skips.

In this paper, we propose a quality management policy based on refined average execution times. These execution times are computed online depending on the input parameters of the corresponding actions. Since dependencies between execution times and input parameters can be very complicated, figuring out a relationship between execution times and inputs without executing the application software on the platform is often untractable for multimedia applications. We use neuronal networks for computing refined average execution times.

3. Neural Networks

Neural networks are well suited to model complex relationships between inputs and outputs or to find patterns in data. Moreover, neural networks are tolerant to noisy data and errors in sample values. In most cases a neural network is an adaptive system that changes its behavior based on information about its environment during the learning phase. In the rest of the paper, neural networks are used for online computation of refined average execution times of the actions. Refine average execution times depend on input parameters of the actions, and thus overcome limitations of standard (fixed) average execution times. This allows better estimation of actual execution times and a reduction of fluctuations of CPU load.

A *neural network* is composed of *layers* (input layer, hidden layers, output layer), each layer is composed of *cells* (or *neurons*), connected to each other by links that are affected by weight. A neuron has an entry, allowing it to receive information from other cells, an activation function. Typical activation functions are $identity : x \mapsto x$ and $sigmoid : x \mapsto \frac{1}{1+e^{-\beta x}}$. Each neuron computes the activation function from the weighted sum of its inputs. The output, can serve as input to other neuron. In Figure 5 we give an example of neural network.

Definition 5 *A neural network is defined by:*
• *a set of layer,* $L = \{ L_1, L_2, \ldots, L_n \}$ *where* L_1 *is the* input *layer,* $L_2, \ldots, L_{n-1}$ *are the* hidden *layers, and* L_n *is the* output *layer.*
• N_l *is the number of neurons in the layer* l.
• $w_{ij}^{(l)}$ *is the weight between the neuron* i *in the layer* l*, and the neuron* j *in the layer* $l-1$ $(2 \leq l \leq n)$.

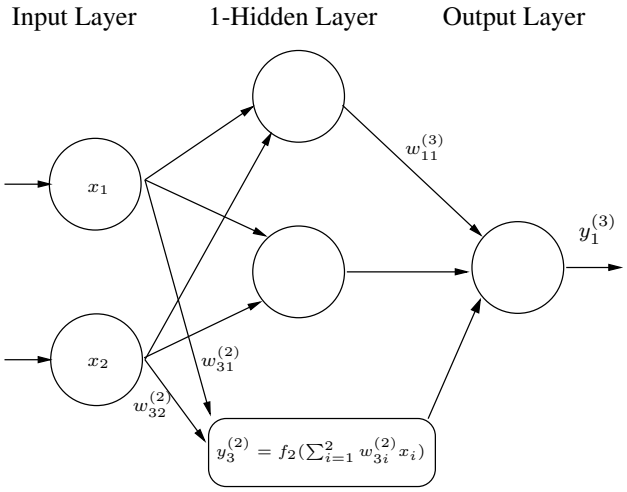

Figure 5. Neural network with three layers

• f_l *is the activation function of the neurons in the layer* l $(2 \leq l \leq n)$.
• $\theta_i^{(l)}$ *is a scalar bias of the neuron* i *in the layer* l.
• $y_i^{(l)}$ *is the output of the neuron* i *in the layer* l.

$$y_i^{(l)} = \begin{cases} f_l(\sum_{j=1}^{N_{l-1}} y_j^{(l-1)} w_{ij}^{(l-1)} + \theta_i^{(l)}) & \text{if } l \neq 1 \\ x_i \text{ (input of neuron } i) & \text{if } l = 1. \end{cases}$$

The design of neural networks that are fitted to a given application is a key issue. Choosing optimal parameters for neural networks is assited by learning algorithms. Parameters that concerns architecture — number of layers, number of neurons in each layers, and activation functions — must be chosen by the designer of the neural network. Weight values between neurons are chosen by learning algorithms, and scalar bias values are often random values.

3.1. Architecture of Neural Network

The choice of architecture for a neural network is essential for an efficient adequation to a given application. Several aspects must be considered when designing the network, the most important are:

- the number of layers (n),
- the number of neurons in each layer (N_l), and
- the activation functions (f_l).

Without hidden layers, the network only offers limited opportunities for adaptation; with one hidden layer, it is capable — with a sufficient number of neurons — to approximate every continuous function [7]. Adding neurons allows taking into account specific profiles of the input neurons. With a large number of neurons, a neural network can stick better to the considered data, but it is not noise-robust.

Once the architecture of a neural network is defined, it remains to choose network weights and scalar bias. Learning algorithms are used to adjust weights between the neurons. We describe a learning algorithm as follows.

3.2. Learning Algorithm

Learning is a process of a neural network design in which the behavior of the network is modified to get the desired one. Neural network learning takes as input a set of *samples*. A sample is a pair (X, Y) where $X = (x_1, \ldots, x_{N_1})$ represents the inputs of the network, and $Y = (y_1, \ldots, y_{N_n})$ are the desired outputs. In the context of this paper, the desired output will be the actual execution time and the output of the network will be the refined average execution times.

During the learning phase, the weights of the network are adjusted in order to match the desired outputs (see Figure 6). This requires a learning algorithm and a set of samples to be learnt. After initializing the network weights W (usually random), and given a sample (X, Y), the learning algorithm computes the error $E(W)$ that measures the difference between the desired output Y and the output of the network for the inputs X. Then, an error correction ΔW is computed and weight corrections are applied. We used backward propagation algorithm, developed especially by [17, 10, 15]. This algorithm is aimed at minimizing the sum of squared differences between the desired output $Y = (y_1, \ldots, y_{N_n})$ and the output of the neural network $y^{(n)} = y_1^{(n)}, \ldots, y_{N_n}^{(n)}$, that is:

$$E(W) = 1/2 \sum_{j=1}^{N_n} (y_j - y_j^{(n)})^2.$$

Network weights are initialized with small random values. If the initial weights happen to be too far from a good solution or if they are near a local minimum of the error function E, the learning will take too many iteration steps for converging to the (global) minimum of E. Furthermore, it may not converge at all to the minimum. For this reason, the method we used for initializing the weights is the same as given in [12]. It is based on the orthogonal least squares algorithm.

After initializing the network weights, a recuring sequence of weights W_i is built in order to approach to the minimum of E. We have $\Delta W_{i-1} = W_i - W_{i-1} = -\epsilon E'(W_{i-1})$, where E' denotes the derivative of E with respect to W, and ϵ is a non-negative real adequatly chosen. Backward propagation algorithm uses the following learning rules:

$$\Delta w_{ik}^{(n)} = -\epsilon \frac{\partial E}{\partial w_{ik}^{(n)}}$$

$$= \quad -\epsilon (y_i^{(n)} - y_i) f_n^{'} (\sum_{j=1}^{N_{n-1}} w_{ij}^{(n-1)} y_j^{(n-1)} - \theta_i^{(n)}) y_k^{(n-1)},$$

and $\Delta w_{ik}^{(p \neq n)} = -\epsilon \frac{\partial E}{\partial w_{ik}^{(p)}} = -\epsilon \delta_k^{(p)} y_h^{(p-1)}$, with:

$$\delta_k^{(p)} = f_p'(\sum_h w_{kh}^{(p)} y_h^{(p-1)}) \sum_{i=1}^{N_{p+1}} w_{ik}^{(p+1)} \delta_i^{(p+1)}.$$

Backward propagation algorithm can be summarized as follows:

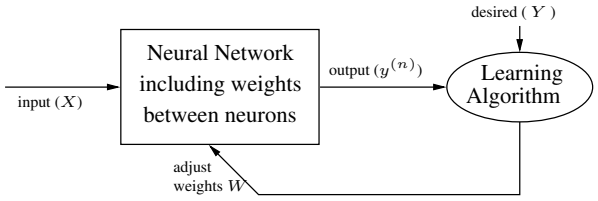

Figure 6. Learning of neural network

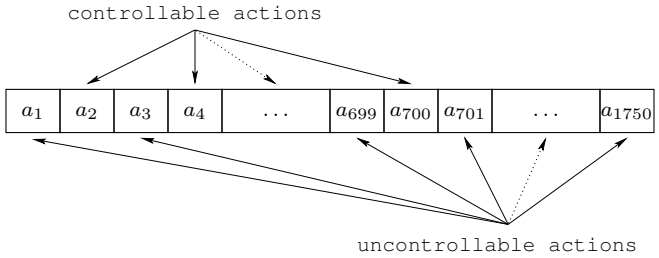

Figure 7. Application software

1. Initialize the network weights W.
2. Select a new sample (X, Y).
3. Update weights of the output and hidden layers using the rules given above, that is, $W \leftarrow W + \Delta W$.
4. Go to 3 if the error $E(W)$ is above a tolerance value.
5. Go to 2 if other samples must be learnt.

Since learning algorithms for neural networks can consume a lot of CPU resources, refined average execution times are learnt offline from execution traces using typical input parameters. This avoid execution time overhead due to learning algorithm.

4. Experimental Results

This section provides experimental results which confirm the interest of theoretical sections. We present the experimental framework as well as a description of the target application (an MPEG4 video encoder) and platform.

4.1. Experimental Framework

We applied our results to an MPEG4 video encoder written in C (more than 10,000 lines of code). The encoder cyclically treats frames. Each frame is split into N macroblocks of 16×16 pixels. Typical values of N range from 180 to $1,620$. In the following, we consider frames of 400×224 pixels ($N = 350$).

The parameterized system $PS(C)$ describing the video encoder consists of:

• the scheduled video encoder, that is, a sequence $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ of $n = 1750$ actions for encoding a frame. This sequence is composed of controllable actions — actions $a_2, a_4, a_6, \ldots, a_{698}, a_{700}$ — and uncontrollable actions — actions a_1, a_3, a_5, …, a_{699}, a_{701}, a_{702}, a_{703},

$\ldots$, a_{1749}, a_{1750} (see Figure 7). Notice that controllable actions are put together at the beginning of the sequence.
• a set of 8 quality levels $Q = \{ 0, \ldots, 7 \}$.

The video encoder architecture is shown in figure 8. The considered application corresponds to a videophone application. It captures a sequence of frames with a camera, transmits the sequence, and then displays the frames on a screen. From a captured frame, the video encoder produces a corresponding bitstream. The latter is transmitted to a video decoder which decodes the bitstream and displays decoded frame on a screen. This architecture uses input and output buffers of the same size K, to cope with changes of load and avoid as much as possible frame skips. These may happen when the input buffer is full.

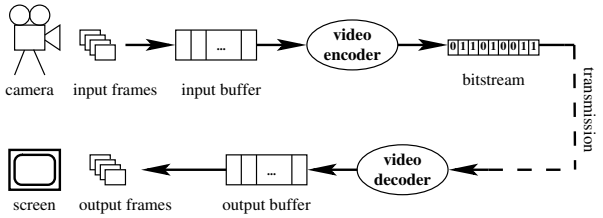

Figure 8. MPEG4 video encoder architecture

We developed a prototype tool that allows the generation of controlled application software (see Figure 1). The inputs of the tool is a parameterized system $PS(C)$:
• the application software $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ modelling actions (C functions) and execution order,
• the set of quality level parameters Q,
• average execution time function C^{av},
• the deadline function D.

From these inputs, the tool computes:
• C code corresponding to the application software,
• tables containing pre-computed values used by the Quality Manager.

A compiler is used to link the following items and generate the controlled application software from:
• the application software and the tables generated by the tool,
• the code for the actions of the application software,
• a generic Quality Manager implementing the stochastic quality management policy.

For the experimental results, the target platform is an STm8010 board from STMicroelectronics. It includes three ST231 processors running at 400 MHz, and it is used in set-top box products. As our approach targets mono-processor platforms, we use only one of the three ST231. A register that counts the number of processor cycles elapsed provides a real-time clock with minimal access overhead.

4.2. Neural Network Architecture

We use neural networks for computing refined average estimates of execution time of actions $a_{701}, \ldots, a_{1750}$. As inputs of these actions can be known in advance, refined average estimates are used by the Quality Manager to better predict their execution times.

Designing a neural network requires choosing the number of layers, the number of neurons per layers, and the activation functions (see previous section). Usually, the number of neurons of the input (resp. output) layer equals the number of input (resp. output) parameters. As the amount of data transmitted between the actions is too large, we do not consider the whole data as an input of the neural network. We only consider $X = (x_1, x_2)$, where x_1 is the SAD value (sum of absolute difference) of the macroblock, and x_2 is the position of the macroblock in a frame. These inputs are sufficient for accurate estimates of execution times. The output layer consists of a single neuron that output the refined average execution time.

There is no general approaches for choosing the number of hidden layers and the number of neurons for each hiddent layer. Literature shows only specific solutions for each problem to be solved. Solutions are seared by means of empirical testing. Usually it is recommended to start with only one hidden layer, and if the results are not acceptable, the number of hidden layers have to be increased [8]. In our case, we remark that one layer is sufficient, and we choose 4 neurons for this hidden layer.

4.3. Results

We provide results for the controlled video encoder running with mixed quality management policy, average quality management policy using fixed average execution times, and average quality management policy using refined average execution times computed by the neural network given above.

Figure 9 shows time budget utilization for a sequence of 140 frames and a single deadline $D = 100\ ms$. The latter corresponds to a frame rate of 10 frame/s. We also take an input buffer size $K = 2$. This means that a frame is skipped if the video encoder complete the encoding of a frame $2D = 200\ ms$ after it is captured by the camera.

The Quality Manager applying the mixed quality management loose about 20 % of the time budget D. Since there is no controllable action from action a_{701} to a_{1750}, the Quality Manager loose control on the execution times of the application after executing action a_{700}. The lost time budget comes from overestimated worst-case execution times combined with uncontrollabibility of actions $a_{701}, \ldots, a_{1750}$. Meeting the deadline implies keeping a safety margin (i.e. the difference between worst-case and average execution times) with respect to average execution times.

Applying average quality management policy using only (fixed) average execution times avoid overestimation of execution times, but does not garantee meeting the deadlines. As a result, overloads are possible, e.g., from

frame 80 to frame 140. Figure 9 shows three jumps corresponding to three frame skips (i.e. buffer overloads). This reduces video quality since a frame is not encoded when it is skipped.

Using neural networks that computes refined average execution times allows better estimates of execution times. This estimates take into account the execution context of the actions. Fluctuations of the load are reduced when using refined average execution times, which avoids frame skipping for the considered sequence and size of the input buffer.

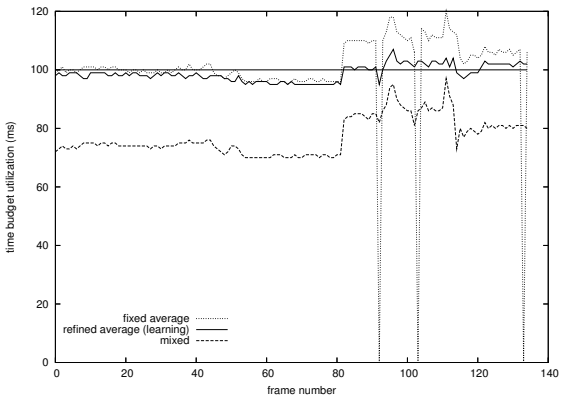

Figure 9. Time budget utilization

5. Conclusion

We presented a fine grain quality management method for real-time applications. The method is based on learning techniques that improve estimates of execution times.

Design methodologies for soft real-time are usually based on average case analysis and do not address predictability of system behavior. In contrast, non flexibility of hard real-time approaches (i.e. based worst-case analysis of system dynamics) often leads to non-optimal use of available resources. Non-optimality comes from considering the (overestimated) worst-case scenario.

Our approach allows a significant reduction of fluctuations of the load for an MPEG4 video encoder, and thus allows a significant reduction of the amount of skipped frames. This improves video quality and this is achieved by the combined use of:

• a Quality Manager controlling the execution times of controllable actions, by selecting adequate quality level parameters, that is, quality levels meeting the quality management policy proposed in this paper;

• neural networks ensuring an early and accurate prediction of execution times of uncontrollable actions.

We work in other directions to improve the method and the supporting tools: adaptation to multiple tasks, considering multi-processor architectures.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for enhancing the flexibility and utility of hard real-time systems. In *Real-Time Systems Symposium*, pages 12–21. IEEE, 1994.

[2] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *RTSS*, pages 286–295, 1998.

[3] J. Combaz, J. Fernandez, T. Lepley, and J. Sifakis. Fine grain QoS control for multimedia application software. In *Design, Automation and Test in Europe (DATE'05) Volume 2*, pages 1038–1043, 2005.

[4] J. Combaz, J.-C. Fernandez, T. Lepley, and J. Sifakis. QoS Control for Optimality and Safety. In *Proceedings of the 5th Conference on Embedded Software*, September 2005.

[5] J. Combaz, J.-C. Fernandez, J. Sifakis, and L. Strus. Using speed diagrams for symbolic quality management. In *IPDPS*, pages 1–8. IEEE, 2007.

[6] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceeding of the IEEE Real-Time Systems Symposium*, pages 222–231, 1993.

[7] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, 1991.

[8] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, 1989.

[9] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. Technical Report TR1996-715, 1996.

[10] Y. LeCun. "une procédure d'apprentissage pour réseau à seuil asymétrique". *Proceedings of Cognitiva 85*, pages 590–604, 1985.

[11] J. Lehoczky and S.Thuel. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the IEEE Real-Time System Symposium*, 1994.

[12] M. Lehtokangas, J. Saarinen, K. Kaski, and P. Huuhtanen. Initializing weights of a multilayer perceptron network by using the orthogonal least squares algorithm. *Neural Comput.*, 7(5):982–999, 1995.

[13] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithm. *special issue of RT Systems Journal on Control-Theoric Approach To Real-TIme Computing*, 23(1/2):85–88, 2002.

[14] L. Papalau, C. M. O. Pérez, and L. Steffens. In S. Goddard, editor, *Work-In-Progress Session of the 16th Euromicro Conference on Real-Time Systems*, pages 33–36, 2004.

[15] D. E. Rumelhart, Hinton, G. E., and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[16] G. M. Schuster, G. Melnikov, and A. K. Katsaggelos. A review of the minimum maximum criterion for optimal bit allocation among dependent quantizers. *IEEE Transactions on Multimedia*, 1(1):3–17, 1999.

[17] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Committee on Appl. Math., Harvard Univ., Cambridge, MA, Nov. 1974.

[18] C. C. Wüst, L. Steffens, R. J. Bril, and W. F. Verhaegh. QoS control strategies for high-quality video processing. In *Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE, 2004.