PoGo: An Application-Specific Adaptive Energy Minimisation Approach for Embedded Systems

Luis Alfonso Maeda-Nunez, Anup Das, Rishad A. Shafik, Geoff V. Merrett, Bashir M. Al-Hashimi
School of Electronics and Computer Science
University of Southampton, UK SO17 1BJ
{lm15g10,a.k.das,rishad.shafik,gvm,bmah}@ecs.soton.ac.uk

ABSTRACT

High performance demand coupled with the need for real-time support, have proliferated the widespread use of battery-operated embedded devices, comprising of one or more processors, across consumer, automotive and commercial applications. System software (such as the operating system) for these devices offers a low-overhead interface to change the CPU voltage and frequency dynamically, satisfying a given performance requirement. This paper proposes PoGo, an approach for energy minimization of embedded systems. Contrary to existing approaches, which are performance requirement-agnostic, PoGo adapts to application-specific performance requirements dynamically, and proactively selects the state that fulfils these requirements while consuming the least power. Proactiveness is achieved by using an Adaptive Exponential Weighted Moving Average (AEWMA) algorithm that adapts to the selected power state. These adaptations are facilitated using a model-free reinforcement learning algorithm. For demonstration purposes PoGo is implemented as a Linux Governor, interfacing with the application and hardware to select an appropriate voltage-frequency control for the executing application. The performance of PoGo is demonstrated on the BeagleBoard-xM, which contains a Texas Instruments' SoC featuring an ARM Cortex-A8 processor. Experiments conducted with multimedia applications demonstrate that PoGo minimizes energy consumption by up to 30% for dynamic workloads and 60% for static workloads as compared to the existing approaches.

1. INTRODUCTION

In recent years the demand for portable battery-operated devices has increased. The high performance requirement for these devices, added to the limited energy supply, makes performance-aware energy optimization a challenging design objective [3]. Of the different components of an embedded system, the microprocessor (CPU) consumes a significant fraction of the total energy and, therefore, lends itself as a primary target for energy optimization. Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) are two hardware techniques for reducing power consumption in the CPU, the former reducing performance together with frequency and voltage, and the latter shutting down unused cores. These techniques rely on control by the Operating System (OS) or the software. OS vendors provide interfaces, enabling developers to control DVFS and DPM; an example is the Advanced Configuration and Power Interface (ACPI).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Different control mechanisms for power management have been proposed in the literature. These studies can be classified into two categories: performance requirement-agnostic and performance requirement-aware. The former approaches are driven solely by energy savings achieved and hardware utilization, so the application's performance requirements are not incorporated in the optimization algorithms; whereas in the latter, the application provides its timing constraints. Dhiman et al. [7, 8] propose a requirement-agnostic power management algorithm that reads the hardware performance monitors (Clock Cycles, Cache Misses, etc.) to measure "CPU intensiveness". Based on an energy-performance trade-off set by the user, the underlying control algorithm adjusts the power management experts, or Voltage and Frequency (V-F) pairs accordingly. The `Ondemand` governor [13] is a popular Linux governor that reacts to current processor workload to adjust V-F. This governor optimises energy to achieve a target idleness; however, if the performance of an application changes within the current execution, either energy or performance are compromised.

Another limitation of these approaches is that they are reactive; decisions are taken after the change in workload has been detected, and thus more cycles are spent in a power state that is not necessarily optimal. A common approach for system-level power management to overcome this limitation is to use workload prediction [2, 6, 9, 14, 15]. A survey of different workload prediction schemes is presented in [14], highlighting the benefits of using adaptive filters. Workload predictions are performed at a coarse-grained interval (5 seconds). Although this approach is an example of proactive power management, it cannot be used for fine-grained prediction due to the lag in the filter technique. Therefore, this limits its use in video and other dynamic applications, where workload changes occur in a much shorter time span [6, 9, 15]. An Exponential Weighted Moving Average (EWMA) neuromorphic controller for workload prediction is presented in [15], implemented as a hardware module, collecting performance readings every $0.2s$. This technique provides higher accuracy for energy efficiency, but suffers from the application performance requirement agnostic nature as discussed before.

Recently, significant studies have been conducted for performance requirement-aware applications. These studies show that frame-based applications allow easy integration of performance constraints to the power management algorithm, achieving real-time performance. Frame processing time is specified as a *deadline*, the inverse of which gives the *frames per second*. The approach of [4] uses experts (similar to [7]), but applied to real-time systems. The technique uses DVFS and DPM together to consider a task's worst-case execution time and deadline, making the algorithm a deterministic selection of the power states. Soft real-time systems, however, need a deadline to adjust and schedule their workload, but this can be occasionally missed, degrading the quality of experience (non catastrophic). Frame-based applications, for example multimedia processing, are considered as soft real-time, as the loss in performance results in a lower frame rate. Choi et al. [6] presents a workload prediction-based power manager for video processors using EWMA. This approach provides high energy efficiency, but is specific only to video decoding. Gu et al. [9] presents a control algorithm for DVFS using frame workload prediction for video games. This is implemented as a Windows Application Programming Interface (API). The approach in [2] improves [6] by using Kalman Filters. Thus, all existing approaches lack a general framework that works uniformly across applications. To address this limitation, we present **PoGo**, a unified power management scheme that supports multiple applications whilst providing energy efficiency and delivering the required performance.

The contributions of this paper are:

- a reinforcement learning-based approach to control the voltage and frequency of the processing cores, specific to the application;
- an AEWMA-based prediction algorithm to forecast workload variations; and
- a thorough validation of the approach through its implementation as a Linux governor, together with an API allowing applications to pass performance requirements and annotations to the governor.

2. DESIGN

In this section the design of PoGo the Run-time Manager (RTM) is described, which involves workload characterization together with the appropriate V-F selection. Figure 1 shows PoGo as a cross-layer framework, interacting with the application, OS and hardware. The communication between layers is indicated by arrows.

2.1 Run-Time Manager

As discussed in Section 1, real-time applications need to complete the execution of a workload (CPU Cycles) within a predefined deadline. The power minimization objective for these applications translates into the solution to a constrained optimization problem. To provide an effective solution to this problem, there are two requirements to be fulfilled: 1) the workload needs to be known prior to its processing such that it can be performed at the lowest V-F value, and 2) once the workload is known (to a certain extent), decisions on the power state have to be taken in such a way that they fulfil the constraint but take into account performance variation of the application.

To address these requirements, we present **PoGo**, a RTM that resides in a general purpose OS. To achieve the first objective, PoGo predicts the next workload for a frame using AEWMA, while for the second objective, PoGo uses Q-learning, an algorithm of reinforcement learning.

The algorithm for PoGo is shown in Algorithm 1. For every new frame, PoGo first predicts the workload, based on this it selects a V-F value. After processing the frame, the performance is determined to fine tune the prediction and the decision algorithms (in their respective Units). The two key steps of PoGo, prediction and decision making are discussed next.

Algorithm 1 PoGo Power Manager

```
 1: Prediction_Unit.Initialise($n$ WorkloadTypes)
 2: Decision_Unit.Initialise(WorkloadRequirement)
 3: for every New Epoch do
 4:     Prediction_Unit.PredictWorkload(WorkloadType)
 5:     Decision_Unit.MapWorkload(PredictedWorkload)
 6:     Decision_Unit.SelectPowerState(V-F)
 7:     Wait until end of frame
 8:     Prediction_Unit.UpdatePrediction(PredictionError)
 9:     Decision_Unit.UpdateQ-Table(TimingError)
10: end for
```

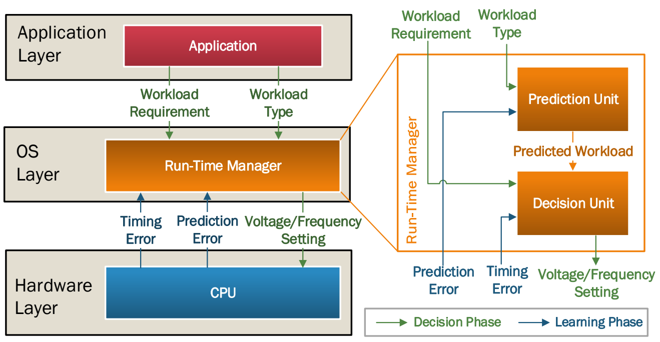


Figure 1. Run-Time Management Unit in the cross-layer approach

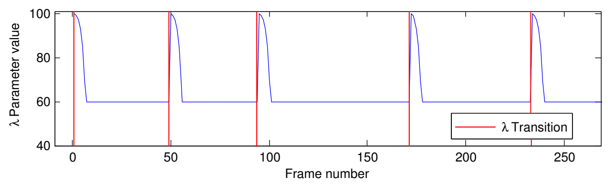


Figure 2. AEWMA λ parameter change at transitions

2.2 Prediction Unit

The Prediction Unit estimates the workload for the next frame using a modified form of EWMA. The EWMA algorithm is widely used in the literature [6, 14, 15] because of its lightweight implementation. The predictor works as an infinite impulse response filter that generates a prediction of the future value based on the average of the previous values weighted exponentially, where the most recent values have greater weights than the older ones. This is shown as:

$$w(n+1) = w(n) \cdot \lambda + \bar{w} \cdot (1-\lambda) \text{ where } 0 \leq \lambda \leq 1 \qquad (1)$$

where $w(n)$ is the current workload at time instance n measured from the hardware, $\bar{w}$ is the average workload in the time interval 0 to n, $w(n+1)$ is the predicted workload at time $n+1$, and λ is the weighting factor. After the prediction has been set, the mean $\bar{w}(n)$ is updated with the prediction according to:

$$\bar{w} = w(n+1) \qquad (2)$$

The parameter that controls the relevance of the past history is the prediction weight λ. At a high λ, recent history data is weighted more heavily than older history, and this helps EWMA to react quickly to changes, but it becomes volatile for random fluctuations. So as the parameter λ decreases, the older history data becomes more relevant, smoothing local variations, reacting slower to changes [5].

In multimedia and other dynamic applications a substantial transition can be observed [12]. The prediction error increases at the beginning of these transitions. The traditional EWMA algorithm is modified to take these transitions into account. This new prediction approach is called the AEWMA. Based on the work by Nembhard [12], once a transition in the workload is detected, the parameter λ is increased to make recent history more relevant. λ is subsequently adjusted to its initial value using an exponential decay function. Figure 2 shows the modification of λ on an application with 4 transitions. The selection of the initial value for λ is explored and justified in Section 4.1.

Another modification to this filter is performed, where frames (workloads) of the same type are grouped together, so predictions are performed on workloads of the same type[1] e.g., for type 1, $w_1(n+1)$ is predicted with $w_1(n)$ and $\bar{w}_1$. Thus, for M different frame types, there are M different predictions, implying that in order to predict the next workload,

[1]For a video processing, workload type translates to the frame type i.e., I, P and B frames.

the predictor requires information of the workload type and the last workload (of the same type). The workload prediction error is dependent on the choice of λ, which is dependent on the application. To improve the prediction accuracy, the prediction unit reads back the hardware performance counters to adjust the prediction weight λ. Note that this filter is very lightweight not only in number of operations per epoch, but in its memory usage, as $\bar{w}$ contains the previous information for that particular workload type.

2.3 Decision Unit

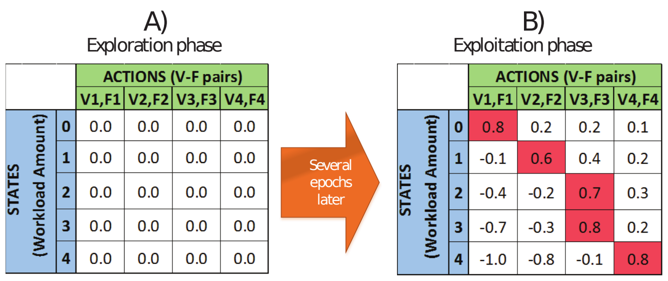
Once the workload for the future frame is predicted, the Decision Unit selects a V-F pair to execute it. This selection is based on the performance constraint given by the application. The decision unit uses Q-Learning (Reinforcement Learning), and builds the model of the system online. The predicted workload corresponds exclusively to the application that communicates with the RTM, and does not include the system-software overheads and other application loads in the prediction. Thus, V-F pairs cannot be directly mapped to a predicted workload using a deterministic algorithm.

The objective of Reinforcement Learning (RL) is to learn to make better decisions under variations. Decisions in reinforcement learning terminology are known as an actions, and the environment is known as states. Originally there is no knowledge of the system, so the decision unit must start exploring decisions in different states to find the optimal (or most suitable) action for a particular chosen state. This is called the *Exploration phase.* Exploration is done by taking a random action for a selected state. Good actions are rewarded and bad actions are penalized. Actions in this context, are the V-F pairs, and states are the different amounts of workload the system may have. It is important to note that the V-F pairs are discrete, so the *best* decision may not be optimal, but it is the best among the V-F pairs available. As an example, let the optimal frequency for a given workload be 533.35MHz; if the CPU supports only 300MHz, 600MHz, 800MHz and 1GHz, the *best* decision is to execute the workload 600MHz. The 'best' in the context of this paper is defined as the lowest V-F pair that fulfills the performance requirement.

Learning is stored as values in a Q-Table, which is a lookup table with values corresponding to all State-Action pairs. At each decision epoch[2], the decision taken for the last frame is evaluated; the reward or penalty computed is added to the corresponding Q-Table entry, thereby gaining experience on the decision. The rate at which actions are rewarded in the Q-Table is determined by the Learning Rate, α, which determines the relative importance of older decisions compared to the newer ones. Initially, the decisions of the algorithm are not optimal. However, with time (after several epochs), the confidence in the selected action improves and the algorithm always selects the best action in a given state. This phase of the algorithm is called the *Exploitation phase.* Figure 3 shows the evolution of the Q-Table. Initially, the values in the Q-Table are all zeros (Figure 3(A)); subsequently, in the exploitation phase, the 'best' actions are determined (highlighted in red in Figure 3(B)).

The transition from exploration to exploitation is not immediate, but is a gradual change, defined as the ϵ-greedy strategy, in which the exploration-exploitation ratio (ϵ) is gradually increased to reduce the random decisions in favor of appropriate decisions[3]. The availability of ϵ makes 're-learning' a feasible operation, especially for dynamic systems in which the best Action for a particular State may change gradually. If relearning is needed, the ϵ may be reduced to allow for more exploration to take place.

[2]In reinforcement learning terminology, the interval at which the algorithm is triggered is known as decision epoch.
[3]Appropriate decisions are those that reduce the energy consumption, while satisfying the performance.


A)
Exploration phase
Several epochs later
B)
Exploitation phase

A) Exploration phase

STATES (Workload Amount)	ACTIONS (V-F pairs) V1,F1	V2,F2	V3,F3	V4,F4
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

B) Exploitation phase

STATES (Workload Amount)	ACTIONS (V-F pairs) V1,F1	V2,F2	V3,F3	V4,F4
0	0.8	0.2	0.2	0.1
1	-0.1	0.6	0.4	0.2
2	-0.4	-0.2	0.7	0.3
3	-0.7	-0.3	0.8	0.2
4	-1.0	-0.8	-0.1	0.8

Figure 3. Q-Table during A) exploration and B) exploitation phases. The red boxes represent the best Action for each State.

2.4 Application Programming Interface (API) Design

We implemented PoGo as a low-overhead API that enables the programmer to take control of the RTM from the application. The three signals that PoGo requires from the application are the performance requirement, the task annotations and the Start/Stop signals. In practical terms, these are defined as:

- **Performance requirement:** PoGo requires a deadline per application execution segment, or a constant deadline defined as a frame rate.
- **Task annotations:** In order to make better predictions, the programmer may be able to separate different application segments or to define a particular workload as a 'workload type'.
- **Start/Stop signals:** These are signals that alert the RTM when the application has started its main loop, and when it finishes.

2.5 FFT Case study: Changes to integrate PoGo API

Let us consider the fft application [10] to highlight the changes needed in the application for use with PoGo. The main loop of the application executes 100 times, changing to three different window sizes. The three different window sizes correspond to three different workload types. The application contains a Program Header, which is responsible for flag parsing, parameter definitions, memory allocations, etc. This is followed by the fft Program Main Loop executing the fft computation. Finally, there is the Program Footer, responsible for memory freeing, file saving, etc. This is represented in Figure 4(A).

In order to use PoGo, different sections of the code are annotated, enabling the application to communicate with the RTM. Figure 4(B) shows these annotations. The Program Header is modified to send the performance constraint together with the Start signal. In the Program Main Loop, the 'New Frame' signal is sent to PoGo, which includes the annotation of which 'workload type' is to be processed. After the Main Loop finishes, the Stop signal is sent to alert PoGo to end listening for new frames. The annotations are designed to be minimal and completely unobtrusive to the rest of the application code.

3. IMPLEMENTATION

The implementation of the RTM as a Linux Governor along with the API is highlighted here.

3.1 Run-Time Manager Implementation

PoGo is designed as a POwer GOvernor for Linux, which is a Loadable Kernel Module, a section of code that extends the functionality of the OS. PoGo can be enabled in a similar manner as other Linux governors e.g. Ondemand, Conservative and Performance; for version 3.7.10 of the Linux Kernel.

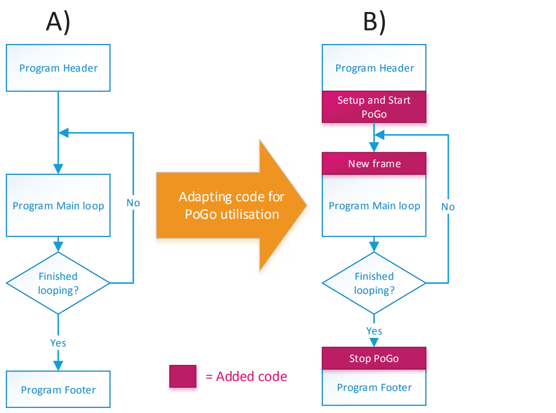
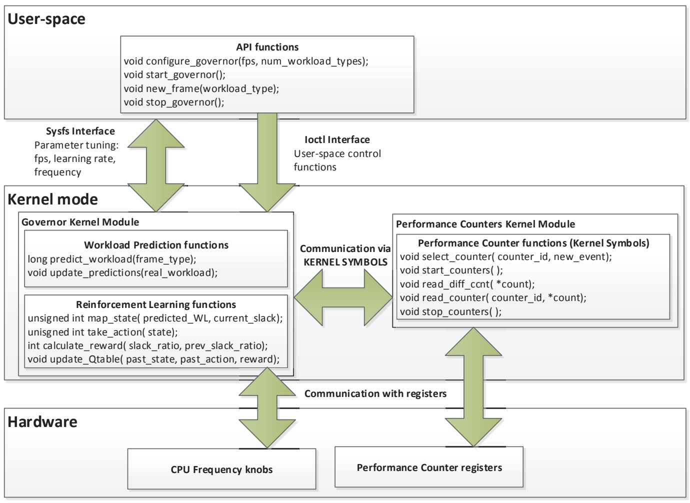


Figure 4. Adaptation of program code for utilisation of the PoGo Run-Time Manager


User-space
Sysfs Interface
Parameter tuning: fps, learning rate, frequency
Ioctl Interface
User-space control functions
Kernel mode
Governor Kernel Module
Communication via KERNEL SYMBOLS
Performance Counters Kernel Module
Communication with registers
Hardware
CPU Frequency knobs
Performance Counter registers

```
API functions
void configure_governor(fps, num_workload_types);
void start_governor();
void new_frame(workload_type);
void stop_governor();
```

```
Workload Prediction functions
long predict_workload(frame_type);
void update_predictions(real_workload);
```

```
Reinforcement Learning functions
unsigned int map_state( predicted_WL, current_slack);
unisgned int take_action( state);
int calculate_reward( slack_ratio, prev_slack_ratio);
void update_Qtable( past_state, past_action, reward);
```

```
Performance Counter functions (Kernel Symbols)
void select_counter( counter_id, new_event);
void start_counters( );
void read_diff_ccnt( *count);
void read_counter( counter_id, *count);
void stop_counters( );
```

Figure 5. PoGo governor implementation

Apart from the intrinsic differences of PoGo with the other governors regarding its own functionality, one of the main features of PoGo is its ability of communicating with user mode via an Application Programming Interface (API). This enables an application developer to have control of the governor from the application. Communication using the API is done using a system call named *ioctl*, which enables a link between User-space and Kernel mode. Figure 5 shows the complete implementation of the PoGo governor. The implementation consists of three sections – the governor module, the API and the Performance Counters module.

The governor module selects action based on predicted workload. The two main task of the governor are workload prediction (using AEWMA) and decision making (using Reinforcement Learning). As discussed in Sections 2.2, the predicted workload for the next frame requires the real workload for the previous frame, which comes in the form of performance counters. The Performance Counters module implements an interface accessible from the governor (and User-space) in order to be able to use the available Performance Counter hardware. This hardware normally comes as a coprocessor adjacent to the CPU cores [1]. On the ARM Cortex A8 [1], the System Control Coprocessor contains the System Performance Monitor, which can detect up to 5 different events simultaneously (including a Cycle Counter).

To collect the workload of the currently processed frame, the governor communicates using Kernel Symbols with the Performance Counter module. The governor has access to all functions available on the Performance Counters module, in order to configure which counters to use, and to request a counter reading. In order to change V-F the governor

Power Mode	Frequency	Voltage (V)	Current (mA)	Power (mW)
OPP50	300MHz	0.93	151.62	141.01
OPP100	600MHz	1.10	328.79	361.67
OPP130	800MHz	1.26	490.61	618.17
OPP1G	1GHz	1.35	649.64	877.01

Table 1. DM3730 Specifications (ARM Cortex-A8) [16]

module uses a system call directly to hardware. Physically this call sends a request to the external Power Management Integrated Circuit (PMIC) to change the Voltage and Frequency (V-F) pairs. In Figure 5, the arrow towards the CPU Frequency knobs is bidirectional, because the PMIC responds with a success/failure signal, which in turn alerts the module with the status of the frequency change command. The Sysfs interface shown on Figure 5 is used for parameter tuning (similar to other Linux governors). Note that in order to reduce the governor execution overhead, the use of floating point operations is avoided, as context switching for the Floating-Point Unit (FPU) (between User-space and Kernel mode) is time consuming. Integers are used, and multiplication/divisions are realized using shift operations.

4. EXPERIMENTAL RESULTS

Experiments are conducted on the BeagleBoard-xM (BBxM) embedded platform, which contains a TI OMAP DM3730 [16] SoC with an ARM Cortex-A8 processor. The platform runs Linux Operating System 3.7.10 together with the Ubuntu 12.04 distribution [11]. Table 1 lists the CPU specifications.

4.1 Prediction Unit

As shown in [2], variation in the workload of an application is dependent on the workload "type" i.e., for video processing, frames of the same type present low variations. Therefore, for accurate predictions, the PoGo governor requires the workload type as one of its parameters every new epoch. The AEWMA filter (refer to Section 2.2) starts with a steady-state weight (λ). This is shown in Figure 2 by the blue line starting at the value of 60%. At every transition (indicated in the figure by the red solid lines) , λ is increased to 100% to give all the weight to the current frame only and ignore previous history. Subsequently, the λ value is restored back to its steady-state using an exponential decay of 2^i. In order to use the AEWMA filter, the optimal parameter for the steady-state λ (60% in the figure) is obtained by analyzing different workloads, as shown in Figure 6. The Mean Absolute Percentage Error (MAPE) is plotted for different steady-state values of the parameter λ run with 5 different VGA (640x480) resolution videos of H.264 encoding. The videos represent dynamic workloads, as each frame presents variations of its own. It can be seen that, beyond steady-state $\lambda = 40\%$, the MAPE is reduced below 4%. All five videos are executed for 720 frames (30 seconds for a 24 fps video). For these training sets, a value of $\lambda \geq 60\%$ gives the best result in terms of MAPE. Finally, using this steady-state $\lambda = 60\%$, Figure 7 shows the real workload (red) and the predicted workload (green) for a segment of VGA video.

4.2 Decision Unit

Figure 8 shows the evolution of the Q-values corresponding to four different actions of one of the Q-Table states (state 4). Results are shown for a VGA video executed for 800 frames. As can be seen from the figure, the Q-value for an action changes as the number of frames up to around 500 frames for most actions. This duration is referred to as the Exploration Phase of the algorithm, where Q-values are modified by applying a reward/penalty. Beyond this point, the algorithm transits to the Exploitation Phase, where no further update of Q-values takes place. It is worth noting that the Q-Learning algorithm works by selecting the highest action for a state and therefore, in state 4 of the Q-Table, action 2 (for 800 MHz) is selected in the exploitation phase.

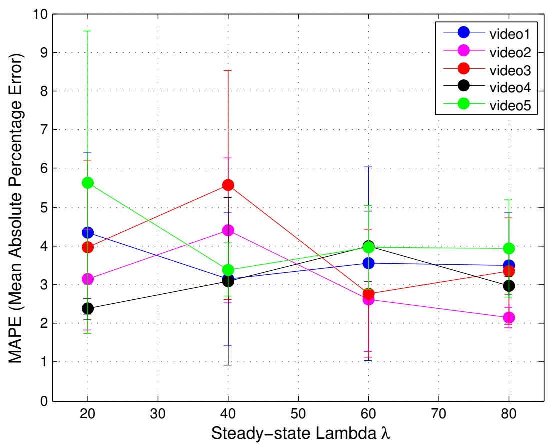


Figure 6. Effect of steady-state weight of λ for AEWMA on prediction error using dynamic workloads

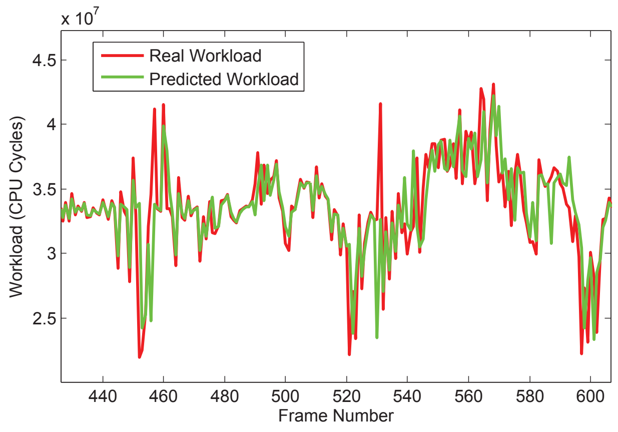


Figure 7. Comparison of real workload vs. predicted

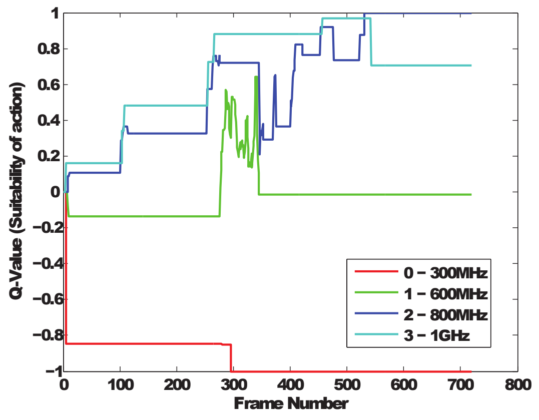


Figure 8. Q-values for different actions in state 4

4.3 Case study: Run-Time Manager For Video Decoding

As mentioned on Section 3.1, PoGo is implemented as a CPU power governor, alongside the other 5 available governors – *Performance*, *Powersave*, *Ondemand*, *Conservative* and *Userspace* (600 and 800) as described below.

- **Powersave:** a static governor (does not change frequency over time) that sets the CPU at lowest possible frequency (300MHz for the BeagleBoard).

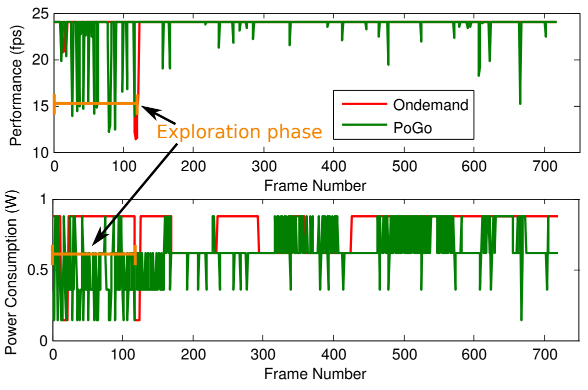


Figure 9. Performance and Power Consumption of the Governors PoGo and Ondemand for an H.264 Video

- **Userspace600:** static, sets the CPU at 600MHz.
- **Userspace800:** static, sets the CPU at 800MHz.
- **Performance:** static, sets the CPU at maximum frequency (1GHz). Performance consumes the most power.
- **Ondemand/Conservative:** dynamic governors that vary CPU frequency based on CPU idleness. The two governors differ from one another in the selection of the frequency steps.

To demonstrate the power and performance trade-off of PoGo, the application *Mplayer* is selected, which uses the *FFmpeg* library of video codecs. The application is modified to include the performance constraint as well as the task annotations as discussed in Section 2.5. For this experiment two codecs are tested – H.264 and MPEG4 decoders. Results are presented for five videos per codec, with each video decoded for 720 frames. This corresponds to 30 seconds of video playback at 23.98 frames-per-second (fps). Each video is composed of 3 different frame types – I, P and B, which represent different *workload types* for the governor. Energy and power consumption are estimated using values from the BBxM datasheet, summarised in Table 1. The power values for the A8 core are reported by running the Dhrystone workloads. An example computation is provided: for a frame with the CPU frequency set at 1GHz, the power consumption for that frame is 877.01mW (row 5, column 5).

Figure 9 compares the PoGo and Ondemand governors in terms of performance and power consumption. Good performance, in this case can be considered by its ability to deliver the target fps for a particular video, which in this case is 23.98 fps. The video decoder decodes frames in real time with a one frame buffer, therefore only one frame is decoded and displayed in a single epoch. This implies that the slack is not accumulative, and is from the decoded frame only i.e., if a frame is decoded in more than 41.7ms (1/23.98), there is a glitch in the video (giving rise to performance loss). Power consumption is measured in Watts (W), which represents the amount of instantaneous power used during a particular frame decoding, therefore total energy consumption is the area below the curves. It can be seen from the figure that the total energy consumption of PoGo is lower than Ondemand, except during the frame intervals 13-22 and 118-124 (both these intervals are part of the learning phase). The learning phase (in orange) shows the period where PoGo is learning the optimal power states for these workloads, by taking random decisions. At around 125 frames, PoGo is able to identify optimal actions and therefore starts taking better decisions. After this learning phase, PoGo enters into exploitation mode, "exploiting" the most adequate decision for every situation. As the workload is dynamic in nature, PoGo continues to explore (sporadically) even in the exploitation phase, resulting in a small performance penalisation.

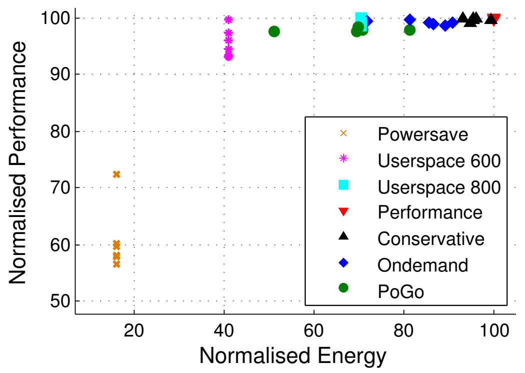


Figure 10. Pareto graph comparing different Governors vs. PoGo by means of energy and performance for H.264 workloads

The Figure 10 plots the Pareto graph of the H.264 codec running 5 videos (represented as a point in the figure). Energy and performance are both normalized: a performance of 100% implies the the video is running at maximum frame rate. Energy is calculated as total energy consumption of the video normalized to the highest energy (corresponding to maximum V-F pair). As can be seen, the static governors have constant energy consumption regardless of their performance. The ideal sector in the figure is the top left corner of the Pareto graphs, which corresponds to the lowest energy consumption with highest performance. In Figure 10 it can be seen that PoGo (shown in green circles) performs better than the other dynamic governors (Conservative and Ondemand), as the performance loss is lower. The energy consumption of PoGo is significantly lower as compared to Ondemand (shown in blue squares) and Conservative (shown in magenta stars). It can be noted that a minimum frequency of 800MHz ensures 100% performance and is only achieved by PoGo.

4.4 PoGo on Static Workloads

To demonstrate the adaptability of PoGo to static workload applications, the "fft" workload is considered from the MiBench benchmark[10]. This application is modified using the API for notifying the governor with the performance constraint and start of frames as illustrated in Section 2.4. In order to provide a frame-oriented approach, the application is executed multiple times in a loop, with each loop representing a frame. A total of 700 frames (loops) are executed. In order to test the adaptability of PoGo to changes in the performance constraints, the workload is kept constant and the performance target (objective) is varied by selecting between 8, 10, 12, and 16 fps. Table 2 shows the results of the performance-energy trade-off corresponding to these performance constraints.

The performance constraints can be interpreted as follows: the larger the performance constraint, the stricter the deadline, i.e., the workloads need to be processed in a smaller time. Intuitively, a high performance constraint requires higher frequencies. This can be seen with the static governors, particularly userspace 800, which satisfies the first three performance targets, but fails to achieve 16 fps (refer to Table 2). It can also be seen that for static workloads, a static frequency proves to be optimal for each target. However, the static workload value cannot be known beforehand and therefore, the desired static governor cannot be set prior to fft execution. PoGo, on the other end, identifies the optimal frequency during the exploration phase.

4.5 Overheads

The implementation of the algorithm as a Linux governor has negligible overhead. Running on lowest frequency on the BBxM (300MHz), the algorithm takes $11.5\mu s$ to compute (from the time of the system call to the end of the decision epoch). For this particular board, the maximum frequency change overhead is recorded as $0.42ms$, with a mean overhead of $0.25ms$. This constitutes $\approx 0.6\%$ of the frame decoding time, assuming a 24fps video. In terms of memory usage, the PoGo governor uses $14kB$ of memory and does not use dynamic memory allocation for the Q-Table.

Governor	Objective: 8fps Performance (fps)	Objective: 8fps Normalised Energy	Objective: 10fps Performance (fps)	Objective: 10fps Normalised Energy
powersave	4.5	16	4.5	16
userspace 600	8.0	41	9.1	41
userspace 800	8.0	70	10.0	70
performance	8.0	100	10.0	100
conservative	8.0	98	10.0	98
ondemand	8.0	100	10.0	100
PoGo	7.5	40	9.7	67

Table 2. FFT benchmark [10] performance vs. energy results.

5. CONCLUSION

We have presented PoGo, a Run-time Manager for application-specific dynamic energy minimization of embedded systems. Energy savings of 30% are achieved by predicting the correct workload using AEWMA and reducing the system voltage and frequency using reinforcement learning to adapt to workload and performance variations. Experiments conducted with static and dynamic workloads demonstrate the advantage of PoGo as compared to the existing governors. An API for utilizing PoGo is introduced, allowing applications to be modified to work with performance constraints for power savings. For heterogeneous behaviour, integration of DSP acceleration with PoGo is work in progress.

Acknowledgments

This work was supported in parts by Centro Nacional de Ciencia y Tecnología (CONACYT) of Mexico and by the Engineering and Physical Sciences Research Council Programme Grant, EP/K034448/1. See www.prime-project.org for more information about the PRiME programme.

References

[1] ARM. ARM Cortex-A8 Reference Manual, 2010.

[2] S.-y. Bang, K. Bang, S. Yoon, and E.-y. Chung. Run-Time Adaptive Workload Estimation for Dynamic Voltage Scaling. *IEEE TCAD*, 28(9):1334–1347, Sept. 2009.

[3] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE TVLSI*, 8(3):299–316, June 2000.

[4] M. K. Bhatti, C. Belleudy, and M. Auguin. Hybrid power management in real time embedded systems: an interplay of DVFS and DPM techniques. *RTS*, 47(2):143–162, Jan. 2011.

[5] G. E. P. Box. Understanding Exponential Smoothing – A Simple Way to Forecast Sales and Inventory. *Quality Engineering*, 1990.

[6] K. Choi, W.-C. Cheng, and M. Pedram. Frame-Based Dynamic Voltage and Frequency Scaling for an MPEG Player. *JOLPE*, 1(1):27–43, Apr. 2005.

[7] G. Dhiman and T. Rosing. System-level power management using online learning. *IEEE TCAD*, 28(5):676–689, May 2009.

[8] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *ISLPED*, pages 207–212, New York, New York, USA, 2007. ACM Press.

[9] Y. Gu and S. Chakraborty. Control theory-based DVS for interactive 3D games. In *DAC*, page 740, New York, New York, USA, 2008. ACM Press.

[10] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite The University of Michigan Electrical Engineering and Computer Science. In *Workshop on Workload Characterization*, 2001.

[11] R. C. Nelson. BeagleBoardUbuntu, 2013.

[12] H. B. Nembhard and M. S. Kao. Adaptive Forecast-Based Monitoring for Dynamic Systems. *Technometrics*, 45(3):208–219, Aug. 2003.

[13] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, 2006.

[14] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *VLSI Design*, pages 221–226. IEEE Comput. Soc, 2001.

[15] S. Sinha, J. Suh, B. Bakkaloglu, and Y. Cao. Workload-Aware Neuromorphic Design of the Power Controller. *IEEE JETCAS*, 1(3):381–390, Sept. 2011.

[16] Texas Instruments. AM/DM37x Power Estimation Spreadsheet, 2011.