

UNIVERSITY OF SOUTHAMPTON

System-Level Power Management using Online Machine Learning for Prediction and Adaptation

by

Luis Alfonso Maeda-Nunez

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Physical Sciences and Engineering
Electronics and Computer Science

2016-07-22

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Luis Alfonso Maeda-Nunez

Nowadays embedded devices have the need to be portable, battery powered and high performance. This need for high performance makes power management a matter of critical priority. Power management algorithms exist, but most of the approaches focus on an energy-performance trade-off oblivious to the applications running on the system. Others are application-specific and their solution cannot be applied to other applications.

This work proposes Shepherd, a cross-layer runtime management system for reduction of energy consumption whilst offering soft real-time performance. It is cross-layer because it takes the performance requirements from the application, and learns to adjust the power management knobs to provide the expected performance at the minimum cost of energy. Shepherd is implemented as a Linux governor running at OS level, this layer offers a low-overhead interface to change the CPU voltage and frequency dynamically.

As opposed to the reactive behaviour of Linux Governors, Shepherd adapts to the application-specific performance requirements dynamically, and proactively selects the power state that fulfils these requirements while consuming the least power. Proactiveness is achieved by using AEWMA for adapting to the upcoming workload. These adaptations are facilitated using a model-free reinforcement learning algorithm, that once it learns the optimal decisions it starts exploiting them. This work enables Shepherd to work with different applications. A programming framework was designed to allow programmers to develop their applications to be power-aware, by enabling them to send their performance requirements and annotations to Shepherd and provide the cross-layer soft real-time performance desired.

Shepherd is implemented within the Linux Kernel 3.7.10, interfacing with the application and hardware to select an appropriate voltage-frequency control for the executing application. The performance of Shepherd is demonstrated on an ARM Cortex-A8 processor. Experiments conducted with multimedia applications demonstrate that Shepherd minimises energy consumption by up to 30% against existing Governors. Also, the framework has been used to adapt example applications to work with Shepherd, achieving 60% energy savings compared to the existing approaches.

Contents

Acknowledgements	xvii
1 Introduction	1
1.1 Research Justification	2
1.1.1 Power Consumption	2
1.1.2 Low Power Design Techniques	3
1.1.2.1 Power Gating	4
1.1.2.2 Dynamic Voltage and Frequency Scaling	5
1.1.3 Challenges of System Level Power Management	6
1.2 Research Questions	9
1.3 Research Contributions	9
1.4 Research Outputs	10
1.5 Document Outline	10
2 Power Management in Microprocessors	11
2.1 Power consumption in Microprocessors	13
2.1.1 Dynamic Power	14
2.1.1.1 Switching Power	15
2.1.1.2 Internal Power	16
2.1.2 Static Power	17
2.2 Power Management Techniques: Knobs	18
2.2.1 Multi-Vdd	18
2.2.2 Dynamic Voltage and Frequency Scaling (DVFS)	18
2.2.3 Power Gating (PG)	19
2.3 Applications and OS: Requirements	22
2.3.1 Real-Time Systems	22
2.3.2 Hard Real-Time versus Soft Real-Time	23
2.4 Power Management: Control	24
2.4.1 Dynamic Power Management (DPM)	24
2.4.1.1 Predictive Techniques	24
2.4.1.2 Stochastic Techniques	26
2.4.1.3 Machine Learning Techniques	27
2.4.1.4 Reinforcement Learning for DPM	29
2.4.1.5 Other Predictive Techniques	30
2.4.2 DVFS	31
2.4.2.1 Workload Detection	31
2.4.2.2 Online Learning	32

2.4.2.3	Offline Learning	32
2.4.2.4	Workload Prediction	34
2.4.2.5	Deadline Prediction	34
2.4.2.6	Application-specific DVFS	35
2.4.2.7	General Purpose DVFS	36
2.4.3	The Interplay of DPM and DVFS	37
2.4.4	Reinforcement Learning	38
2.5	Discussion	41
3	System-Level Power Management	43
3.1	Run-Time Management	43
3.1.1	Requiriements	44
3.1.1.1	Application adaptation for use with RTM	47
3.1.2	Power Knobs	47
3.1.3	Control	48
3.1.3.1	Monitor Feedback	48
3.1.4	Run-Time Management Algorithm	48
3.2	Prediction Unit	50
3.2.1	Motivation	50
3.2.2	EWMA	51
3.2.2.1	Optimisation and Results	52
3.2.3	AEWMA	54
3.3	Decision Unit	57
3.3.1	Cost Function	58
3.3.2	Learning Phase in Shepherd	60
3.4	Discussion and Summary	61
4	Implementation of Run-time Manager	65
4.1	Implementation as Linux Governor	65
4.1.1	Restrictions and optimisations for Linux Governor Implementation	67
4.2	Implementation of Prediction Unit	68
4.3	Implementation of Decision Unit	70
4.3.1	Implementation of Learning Phase in Shepherd	71
4.3.2	Action Suitability for Exploiting Learning	73
4.4	Performance Counters Module	74
4.5	An Application Programming Interface (API) for Shepherd	75
4.5.1	Design flow for application adapted to use Shepherd Governor . .	77
4.6	Discussion	78
5	Results	79
5.1	Experimental Setup	79
5.2	Case Study: Run-Time Manager for Video Decoding	81
5.2.1	Real time behaviour of Shepherd	82
5.2.1.1	Shepherd on H.264 VGA	84
5.2.1.2	Shepherd on H.264 QVGA	85
5.2.1.3	Run-time Manager (RTM) on H.263	86
5.2.1.4	Shepherd on MPEG2	86

5.2.1.5	Shepherd on MPEG4	87
5.2.2	Governor comparison	87
5.2.2.1	Comparison versus Dynamic Governors	87
5.2.2.2	Comparison versus Static Governors	88
5.3	Shepherd API results	89
5.4	Overheads	91
5.5	Discussion and Summary	92
6	Conclusions	93
6.1	Future Work	94
	Bibliography	97
	Appendix A - Sample Shepherd Output Files	107
	Appendix B - Publications	109

List of Figures

1.1	Dynamic and Leakage Power trend[1]	3
1.2	Power vs Energy. Reproduced from [2]	6
2.1	Flowchart of an embedded system in a cross-layer approach. Analytical approach (left) and practical approach (right). Sections 2.2, 2.3 and 2.4 are described as per the figure	12
2.2	A CMOS Inverter showing the charging and discharging of the capacitor C_{load} . Based on [3]	15
2.3	A CMOS Inverter showing the short circuit current flowing from V_{DD} to V_{SS} . Based on [4]	16
2.4	Leakage Currents in a CMOS circuit (a). Taken from [2]. ITRS trends for leakage power dissipation (b). Based on [5].	18
2.5	Definitions of power and times for a device with IDLE and SLEEP modes	20
2.6	Markov chain and Markov Decision Process model for DPM of StrongARM SA-1100, as presented by Benini et al. [6]	26
2.7	Decision Tree example for selecting V-F settings. Taken from [7].	33
2.8	Reinforcement Learning simple diagram. Taken from [8]	38
3.1	Generic cross-layer RTM, where the arrows show the communication between the layers.	44
3.2	Run-Time Management Unit in the cross-layer approach	49
3.3	Comparison of real workload vs. predicted on a sample video, from frames 430 to 615	52
3.4	Effect of weight for Exponential Weighted Moving Average (EWMA) on prediction error using dynamic workloads	53
3.5	Effect of weight for EWMA on prediction error using static workloads	54
3.6	Sample of video with frame types 1, 2 and 3. Red line represents the group of frames of type 2 (cross) and 3 (green filled circles) between frames type 1 (blue hollow circles) used to calculate local standard deviation.	55
3.7	AEWMA λ parameter change at transitions	56
3.8	Cost function of Q-Learning algorithm for Shepherd. This graph shows the level of reward/punishment obtained for finishing the workload too early (A), very close to the deadline without surpassing it (B), finishing the workload shortly after the deadline (C) and finishing very late (D). Time left for deadline is calculated as $100 \frac{t_{deadline} - t_{finished}}{t_{deadline}}$	60
3.9	Q-Table during A) exploration and B) exploitation phases. The red boxes represent the best Action for each State.	61
3.10	Suitability of actions for a particular state (state 4), defined by the Q Values of the different actions	62

4.1	Shepherd governor implementation	67
4.2	Random number distribution generated from taking the 8 Least Significant Bits (LSBs) from the CPU cycles measured per frame, over 5760 frames.	68
4.3	Adaptation of program code for utilisation of the Shepherd Run-Time Manager	78
5.1	BeagleBoard-xM[9] development board used for running the experiments of Shepherd	80
5.2	Video Decoding experiment using Shepherd Governor running on BeagleBoard-xM	81
5.3	Performance and Power Consumption of the governors Shepherd and On-demand for an H.264 Video	83
5.4	Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H264 VGA Video	84
5.5	Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H264 QVGA Video	85
5.6	Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H263 Video	86
5.7	Pareto graph comparing different governors vs. Shepherd by means of energy and performance running MPEG2 Video	86
5.8	Pareto graph comparing different governors vs. Shepherd by means of energy and performance running MPEG4 Video	87
5.9	Performance and Power Consumption of the governors Shepherd and On-demand for iFFT benchmark at 10 frames-per-second (fps) constraint . .	91

List of Tables

1.1	Common Low Power Design Techniques for Complementary Metal Oxide Semiconductor (CMOS)	3
2.1	Hard Real Time versus Soft Real Time comparison, as explained by Sally [10]	24
2.2	Symbols of Q-Learning algorithm	39
3.1	Comparison of variation of the workload with and without grouping into frame types	53
3.2	Comparison of variation of the workload with and without grouping into frame types	55
4.1	Q-Values of State 3 of Q-Table from Listing 4.2	74
5.1	DM3730 Specifications (ARM Cortex-A8 processor) [11]	79
5.2	Comparative table showing the paretos of energy and performance averages of every Linux governor compared to Shepherd. Numbers are normalised to 100. Ideal performance is 100, and ideal energy is 0.	88
5.3	FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective	90
5.4	FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective	90
5.5	Inverse FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective	90
5.6	Inverse FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective	91

List of Acronyms

CPU	Central Processing Unit
GPU	Graphics Processing Unit
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
RF	Radio Frequency
CMOS	Complementary Metal Oxide Semiconductor
HW	Hardware
OS	Operating System
IPS	Instructions-Per-Second
WCET	Worst - Case Execution Time
TI	Texas Instruments
BBxM	BeagleBoardxM
SoC	System-on-Chip
SotA	State-of-the-Art
HDD	Hard Disk Drive
RTOS	Real-Time Operating System
GPOS	General Purpose Operating System
RTS	Real-Time System
RT	Real-Time
API	Application Programming Interface
FPU	Floating-Point Unit

SR	Service Requester
SQ	Service Queue
SP	Service Provider
MDP	Markov Decision Process
SIMD	Single-Instruction-Multiple-Data
LSB	Least Significant Bit
PMU	Performance Monitoring Unit
JTAG	Joint Test Action Group
FD	Frame-Dependent
FI	Frame-Independent
V-F	Voltage and Frequency
PM	Power Management
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
DPM	Dynamic Power Management
RTM	Run-time Manager
RTPM	Runtime Power Manager
PG	Power Gating
ACPI	Advanced Configuration and Power Interface
fps	frames-per-second
ML	Machine Learning
RL	Reinforcement Learning
W	Weight
EWMA	Exponential Weighted Moving Average
AEWMA	Adaptive Exponential Weighted Moving Average
MAPE	Mean Absolute Percentage Error
ANN	Artificial Neural Network

FM Frequency Mapper

α Learning Rate

S State

A Action

PC Program Counter

CSV Comma-Separated Values

Acknowledgements

I would like to thank first and foremost my parents, Alfonso and Aida, and my brothers Juan Carlos and Roberto, for their lifelong support in the good and the hardest times. You have taught me the most important things in life, and I would not be where I am without you. Not only am I proud to have you close, but deeply honoured to be a part of this family. I would like to thank also my supervisors Geoff Merrett and Bashir Al-Hashimi, as they have guided me through the long and winding road that is the PhD. I would like to thank my sponsor the National Council of Science and Technology (CONACYT) in Mexico, because without them I would not be able to do a postgraduate programme here. I would like to thank my friends which helped me through the difficult times, Isaac, Mauricio, Eryx, Domenico, Jean-Marie, Vasilis, Jaime. Finally, I would like to thank my colleagues who taught me valuable lessons on the PhD and life itself, Andy, Shida, Sheng, Anup, Rishad, Alex Wood, Matt, Davide, Asieh and Biswas. Thank you all for this wonderful experience.

Chapter 1

Introduction

In recent years the demand for portable devices has increased with requirements of very high performance and multi-functionality contained in relatively small devices. These devices include consumer electronics such as smartphones, tablets, cameras, etc. and reliable medical pervasive systems, in which their portability and mobility suggests that they use a constrained energy source such as a battery. For example, security cameras not only record video, but do object recognition, face detection and other computing intensive algorithms, while streaming the data to a server.

Besides the normal function of being able to make/receive calls, nowadays smartphones are expected to play and record HD video, surf the web, connect to social networks, transfer files wirelessly, provide location services and maps, play music and take pictures in a fast and reliable way, without draining the battery too quickly. The studies by Carroll and Heiser [13, 14] on smartphone power distribution suggest that apart from the antennas and the display, Central Processing Unit (CPU) contributes significantly to power consumption. As technology of smartphones evolves, average power does not increment substantially but maximum power does [14]. Therefore, not only battery life becomes a concern, but temperature increase, potentially reducing overall lifetime of the devices. Mahesri and Vardhan [15] performed a study on energy consumption for laptops, demonstrating that with intensive workloads CPU is the main source of power consumption. In addition, the energy costs are an increasing concern for data centres, these come from powering and cooling them[16]. Hence, the spectrum covering high performance needed in battery powered devices and their limited energy supply, as well as the server power and cooling; make energy efficiency a very important subject to address.

Addressing the energy problem, components of embedded devices have been improved. The efficiency of batteries has increased, providing more capacity, and incremented lifetime. Modern antennas can be powered down when idle. The introduction of multicore technology has reduced the load for a single CPU, allowing for several cores to perform

different tasks at the same time. Another technique for increasing the efficiency of embedded devices has been to distribute computation through the multicores, and to add application specific modules, or hardware accelerators; to the system. These accelerators include Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Floating-Point Units (FPUs), Field-Programmable Gate Arrays (FPGAs) hardware video codecs, camera interfaces, Radio Frequency (RF) transceivers, etc. The traditional CPU is then part of a larger System-on-Chip (SoC) with several internal modules. As mentioned by Carroll and Heiser [14], the use of hardware accelerators such as video decoders and GPUs has helped reduce the load for the CPU. But these technologies alone are not enough to fulfil the performance requirements with an acceptable battery life.

CPUs have been identified as one of the main sources of power consumption. Several technologies have been developed for semiconductors to make energy consumption more efficient. Some of these include the optimisation of the fabrication technology and the implementation of low power design techniques, the design of power efficient memory devices, and the reduction and variation of the power supply voltage and frequency. Section 1.1 provides an insight on these technologies, identifying the need for better optimisation.

1.1 Research Justification

1.1.1 Power Consumption

The CPU of an embedded device is a digital Complementary Metal Oxide Semiconductor (CMOS) circuit that consumes energy to stay active. Modern CPUs are part of a larger SoC, in which the latter includes other components such as GPUs and Hardware (HW) accelerators, each of these components consuming its share of energy. Energy can be defined as the power consumption of the component per unit of time¹ [3]. This dissipated power comes from two sources, *Dynamic Power* (or switching power) and *Leakage Power* (or static power). Dynamic Power dissipation happens when the transistors inside a CMOS circuit are switched, allowing internal capacitances to charge/discharge and short-circuit currents move through the circuit. The Voltage, the load capacitance and the switching frequency determine the amount of Dynamic Power. Leakage Power dissipation does not depend on switching frequency but rather on the fabrication technology used, as the intrinsic parasitic currents of the transistors. As fabrication technology has reduced the size of transistors, Leakage Power has increased, exceeding switching power, as seen on Figure 1.1. Until recently, CPU design was more concerned with performance improvement and silicon area reduction for lower

¹A more detailed explanation on Power Consumption in CMOS is given in Section 2.1.

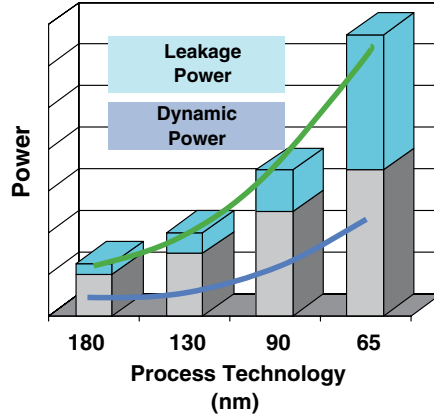


FIGURE 1.1: Dynamic and Leakage Power trend[1]

manufacturing costs, but power is now one of the major design constraints.

$$P(t) = I(t)V(t) \quad (1.1)$$

$$E = \int_0^T P(t)dt \quad (1.2)$$

1.1.2 Low Power Design Techniques

Power minimisation is a major concern which has given rise to the development of power saving techniques. These techniques can be classified depending on the circuit abstraction level, from bottoms-up being: Logic and Technology level, Architecture and Circuit level and System level. These techniques can also be classified depending on whether they target more on either Dynamic or Leakage Power savings. Table 1.1 shows this classification with their target. It is fair to say that the techniques are not exclusive for Dynamic or Leakage, but they tend to save more power on their targets. A thorough explanation of these techniques is given in Section 2.1.

Abstraction Level	Dynamic Power	Leakage Power
System level	DVFS, Multi-Vdd	Power Gating
Architecture and Circuit level	Clock Tree Optimisation, Clock Gating, Operand Isolation	Adaptive Voltage Scaling
Logic and Technology level	Logic Resizing, Logic Restructuring	Multi- V_{th} , Adaptive Body-Bias

TABLE 1.1: Common Low Power Design Techniques for CMOS

Architecture, Circuit, Logic and Technology level techniques are autonomous in the sense that no software control is needed for them to be triggered. System level techniques, however, require a controller for their optimisations to yield higher power savings. This control can be embedded on the Operating System (OS) (e.g. in Real-Time OSs), be developed as a driver (e.g. Linux Governors[17]), or manually set by the user. In any of these situations, this is software driven control. This control affects the performance of the SoC. For example, using Power Gating (PG) in a CPU effectively turns off its clock and voltage, leaving it unable to do any processing whilst powered down. Using Dynamic Voltage and Frequency Scaling (DVFS), the CPU clock and voltage are reduced, decreasing also both the power consumption and the Instructions-Per-Second (IPS), in which the latter can be seen as a reduction in performance. Multi-Vdd[1] is a high level design technique that requires having the system divided in several voltage domains, in which each of them can be either a CPU or a hardware accelerator, and each of them has its own voltage supply and clock. Depending on the application, the different domains can have their power when full performance is not needed from them. Further description of Multi-Vdd is explained in Section 2.2.1. As seen on these last three examples, performance and power are at a trade-off. This has been one of the fields of study since the system level techniques were developed.

1.1.2.1 Power Gating

Power Gating (PG) allows for an SoC to effectively remove the voltage and clock of blocks that are not being used [2]. The strategy is to provide power modes to the blocks, low power mode and active mode. This is a very effective technique because it reduces both dynamic and leakage power. The implementation of PG includes the functional block to be gated, the controller, a power switching fabric, isolating logic and other components that are always on. The controller triggers a sequence that allows the state of the functional block to be saved (for after powering up again), as well as the stopping of the clocks together with the switching of the voltage supply. The saved state (or State Retention[2]) uses registers with reduced leakage power so when the functional block is off, only the always on logic power consumption is considerably low.

The complexity of using PG implicates time and energy required to power down/up the functional block, making it crucial to determine the situations in which to trigger PG for the system to provide the desired performance. The strategies developed to determine when to use PG are known as Dynamic Power Management (DPM). These strategies do not only include power management of the functional blocks inside the SoC, but the control of the low power modes of external peripherals (called in this text *peripheral DPM*). The main concern of DPM is the overhead for powering on/off. This overhead is composed by the higher power needed when the device is powering on/off, and the time it takes for this. Therefore the constraint is to determine whether it is

worth this overhead or not. The break-even time is defined as the minimum time the device needs to stay in low power mode for the energy consumed to be equal to that in the active mode for the same time. Some functional blocks/peripherals have different power states, the deeper sleep states have generally greater overheads [18]. Substantial research [6, 19–28] has been done for DPM to make more efficient decisions with less energy/time penalties. A survey of DPM strategies is presented in Section 2.4.1.

1.1.2.2 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a low power design technique that allows CPUs and other processing elements (DSPs, GPUs) to reduce their performance by lowering their voltage and frequency, therefore decreasing power consumption. The CPU is not shut down by DVFS, and therefore power reduction is not as drastic as with PG. Nonetheless, power savings are achievable when the task at hand does not require full speed of the CPU. The processing elements that use DVFS are powered by a programmable power supply, together with a System Clock (SysClock) Generator that provides a clock signal to the element. In order to run the CPU at a given frequency it will require a minimum voltage to run, and at higher frequencies higher supply voltages are needed. The sequence to use DVFS requires programming both the power supply and the clock generator at runtime[2]. This is further explained in Section 2.2.2.

As DVFS reduces performance when using lower frequencies, its use has to be controlled to allow demanding tasks to execute faster, if the task requires a given performance. DVFS is also used for thermal concerns, as high power consumption yields more heat. The main concern with DVFS is that reducing power consumption does not necessarily reduce energy consumption. Figure 1.2 shows how power is reduced from Approach 1 to Approach 2, but Energy consumption for both approaches (grey area below the curve) may not be necessarily reduced for the same workload. Care should be taken then to ensure that DVFS is used to target energy reduction, not only power. DVFS has also been used to for thermal optimisation [29]. This because reduction in power consumption directly correlates with temperature reduction, which enhances lifetime reliability of the system. Many modern commercial SoCs provide support for a discrete number of voltages and frequencies[2], usually paired together. These will be referred in this document as the Voltage and Frequency (V-F) pairs. The V-F pairs supported are SoC specific.

In order to have access to these system-level techniques, the industry specification called Advanced Configuration and Power Interface (ACPI) has been created[30]. ACPI sets the standard interface needed to facilitate communication of the hardware with the software and the OS. This interface allows for the OS to configure the power states of the SoC, as well as for the SoC to notify its available power levels, both sleep states

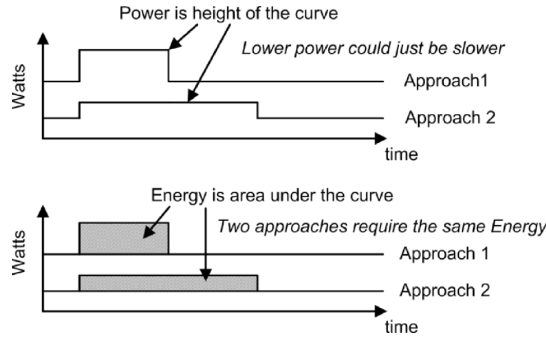


FIGURE 1.2: Power vs Energy. Reproduced from [2]

for DPM and voltage/frequency levels for DVFS. Next section analyses the challenges faced nowadays by the industry and academia to achieve better energy efficiency.

1.1.3 Challenges of System Level Power Management

As mentioned before, System Level Power Management techniques control the power state of the processing elements. Therefore, the main challenges of using these techniques reside on how to efficiently reduce energy consumption without making a significant impact on performance. DPM and DVFS approaches are different in usage, as the strategies to optimise energy consumption are contrasting. The common strategy for DPM is to exploit large idle times, allowing the processing element to sleep more. On the contrary, the common strategy for DVFS is to exploit short idle times by reducing performance, maintaining the processor active at lower power consumption.

Operating Systems (OSs) play a major role in the control and efficiency of power management. Modern OSs are equipped with power controllers that take advantage of these idle times to provide lower energy consumption[17, 18]. Moreover, the scheduler will impact the power efficiency implicitly, as scheduling decisions will affect the amount of idle time between processes. Real-Time Operating Systems (RTOSs) are used for applications that have a strict constraint and therefore need to yield a defined amount of performance within a specific time frame. This creates the need for deterministic behaviour in which the workload and deadline are known beforehand, and therefore can be scheduled accordingly. Power management for RTOSs has been studied before [27, 31], where not only DPM and DVFS have been analysed separately but their interplay, allowing to decide whether to run as quick as possible then sleep or to run at lower frequencies.

The DPM - DVFS interplay can exist inside the same processing element e.g. CPU, with sleep states (DPM) and DVFS, for which the energy saving strategy should be a mix of both. The main challenge is to decide whether there are more energy savings running at maximum speed to reach idle period or to reduce speed given the current

workload. The current workload has to be known a priori. For example if the processing element were a CPU that requires to do intense video decoding, idle times may not make DPM feasible, whereas the DVFS strategies could work better. On the contrary if the application required has long idle times e.g. word processing, the DPM strategies may yield greater energy savings. An example of the interplay of DPM - DVFS is presented by Das et al. [32], where the approach is to decide whether to slowdown (using DVFS) or to run-to-idle (DPM) characterising the applications based on their break-even points (defined in Section 2.2.3).

General Purpose Operating Systems (GPOSs) (non-RTOSs)[33] do not explicitly have RTOS timing constraints, and therefore the approach for power management is targeted differently. The objective most commonly followed is to preserve a performance throughput. This throughput is measured with different metrics, yielding different results. Linux OS has in its kernel options for DVFS called governors, in which the performance objective is given as the idle time percentage of the CPU in periodic intervals, making frequency decisions to reach this target [18] i.e. if measured idle time is lower than the parameter, increase frequency, and vice versa. Some state of the art approaches use performance monitors to characterise the workload, and adjust frequency based on a usage factor calculated from the characterisation[22].

While the previous approaches are efficient for overall performance, they are oblivious to the applications running on the OS, potentially consuming more energy than required or providing lower performance than needed. Some approaches have created application specific algorithms to attack these shortcomings, some focused in video decoding[34, 35] and video games[36] for example. These approaches usually target real-time performance, resembling the behaviour of RTOSs. These approaches obtain performance constraints from the application running and based on the workload select the V-F pairs to keep that constraint. The energy efficiency of the algorithms is increased, but they remain application specific and are unable to run across different applications.

Moreover, the power control system of these approaches is embedded to the specific application, so replicating the code for another application is not trivial and may be time consuming. Added to this complexity, a recent survey [37] demonstrated that software developers have limited knowledge about energy efficiency, lack the knowledge about techniques to reduce energy consumption of software, and do not know how the software is using the energy. Therefore, a framework for programming different applications must exist to allow a power controller to provide energy efficiency across these applications. This framework should allow high level software developers to provide energy efficiency to their applications without the need to understand the complexity of the power management algorithms.

As mentioned before, modern SoCs integrate several modules (or components), in which some of the modules implement DVFS and others PG. Therefore, the strategies

for more efficient energy savings in the SoC depend on the low power hardware of the module. The OMAP processors by Texas Instruments[38] present an environment with both DVFS for some modules and PG for others. The strategy on interplay of both DPM (in Power Gating) and DVFS is different for each module. The PG modules e.g. WI-FI antennas, bus peripherals, memories, etc. can each be managed by analysing their usage rate, and whether or not a complex DPM algorithm may be required to control their power states. The DVFS modules e.g. CPU, DSP, GPU may be controlled using complex DVFS algorithms, with the power states of both PG and DVFS modules being controlled by a central power management system. The development of efficient the DPM and DVFS algorithms for the power management is imperative to provide significant power optimisations.

The challenges for system-level power management can be summarised then as:

- Power management techniques exist (PG, DVFS), and the algorithms to control these need to achieve a target performance.
- The power interfaces to communicate the hardware and software exist (ACPI), and the GPOSs have implemented algorithms to use them.
- State-of-the-Art for power management of GPOS is oblivious of the applications running, while State-of-the-Art for application specific power management is constrained to a single application.
- Software developers need a framework to take advantage of the power management algorithms running at OS level without the need for knowledge of the complexity of the algorithms.

1.2 Research Questions

1. How can embedded devices save energy at system level in an unknown environment where workloads and applications vary, without losing performance?
2. How can this energy saving technique interact with its environment? How can it be used in real life?
3. How does the technique developed perform compared to existing State-of-the-Art?

1.3 Research Contributions

The overall contribution of this research is the proposal for a new framework which optimises power consumption on embedded processors, enabling software developers to create power-aware applications. This contribution is divided into:

1. The design of a cross-layer Runtime Power Manager to optimise power consumption using DVFS embedded processors with an acceptable impact in performance. This Run-time Manager (RTM) uses Machine Learning (ML) optimisation algorithms for prediction and adapting of power states.
2. The design of a power-aware programming framework for:
 - Converting general purpose applications into soft Real-Time (RT)
 - Measuring performance as part of the Run-time Manager
 - Allowing applications to become power-aware by communicating their performance constraints to the Run-time Manager
3. Evaluating the effectiveness of the Run-time Manager on DVFS. This includes the measurement of performance loss, power consumption, overheads, and comparisons with the State-of-the-Art.

1.4 Research Outputs

This research has yielded three publications:

- “PoGo : An Application-Specific Adaptive Energy Minimisation Approach for Embedded Systems” published in HiPEAC 2015 [39]. I was the first author of this paper, doing all theoretical and experimental work and writing the majority of the paper.
- “Towards automatic code generation of run-time power management for embedded systems using formal methods” published in MCSoc 2015 [40] I was the second author of this paper, taking the implementation from the PoGo paper, executing experiments on the real platform, and performing data analysis.
- “Learning transfer-based adaptive energy minimization in embedded systems” published at IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, August 2015 [41] My contributions for this paper include the aid in the literature review and the conceptualisation of the algorithm used.

1.5 Document Outline

An overview of Power Management (PM) technologies has been given in this chapter, with the motivation on the role OSs play in the power saving of the devices. This Thesis is divided into 6 chapters. Chapter 2 gives a description of System-Level Power Management, providing an overview of the existing technologies, as well the State-of-the-Art techniques for optimising them. Chapter 3 proposes a global Run-time Manager (RTM) for Power Management (PM), with the theory behind efficient V-F selections. Chapter 4 describes how the design was developed for a real platform running Linux OS. Chapter 5 analyses the experiments run, highlighting the achievements of the proposed algorithm, and Chapter 6 provides Conclusions and Future Work.

Chapter 2

Power Management in Microprocessors

Power consumption in Complementary Metal Oxide Semiconductor (CMOS) circuits is a major concern nowadays, especially in embedded devices. For this reason, power management techniques have been developed to reduce power consumption by reducing the components' voltage. These techniques are known as power knobs, as they can be tuned, either automatically or manually. These power knobs include Power Gating (PG) and Dynamic Voltage and Frequency Scaling (DVFS), where the former completely shuts down a segment of hardware, and the latter reduces a processor's performance by reducing its frequency.

When addressing general purpose systems normally these run an Operating System (OS), on top of which run the applications. Depending on the context these applications and/or OSs may have performance requirements that the system must meet, whether it is a particular frame rate for video or an amount of computations per second. In other cases the system does not have these constraints, so any performance the system yields will suffice. This may seem well for the system, but there may be an impact in user experience [42]. Therefore, it is logical to take advantage of the opportunities that arise from knowing the performance constraints and the power knobs existing on the system.

It helps to see an embedded system as a cross-layered system. Let the system be a modern mobile phone. Analytically it can be divided into three layers:

- The physical or Hardware layer, where the processor resides, all of the electronic components and the battery. This layer includes the power management hardware it has embedded for reducing power consumption. In a modern mobile phone the hardware may have a quad-core ARM microprocessor, with 3G and Wi-Fi

antennas, battery, touchscreen, hardware accelerators and other peripherals. Each of these components consumes energy, which is provided by the battery (with limited energy).

- The Operating System layer, which is the overall manager of the system, controlling execution, scheduling tasks, memory management. It also controls the power states of the hardware. Depending on the system, the OS may be General Purpose or Real-Time. In the mobile phone example, the OS running may be general purpose e.g. Android, Linux, iOS, Windows Phone.
- The Application layer, where the user applications run. Here, applications running ask the OS for permission to run. Applications vary widely in their purpose, and therefore in their resource use, whether requiring some antennas for communication, intensive use of the processor for calculations, etc. Examples include text processors, video and audio applications, instant messaging, video games, etc.

Given that embedded devices are normally energy constrained, it is necessary to develop strategies to optimise energy consumption in order to extend the use time of the devices battery life. There are many components consuming energy in an embedded device (processor, screen, antennas, hardware accelerators, etc.). This work focuses in the power consumption of semiconductors, specifically of processors, the brain of the system. First, it is imperative to define what is power and energy consumption in a processor. Section 2.1 describes what power consumption is in semiconductors, setting up the need for better power saving techniques.

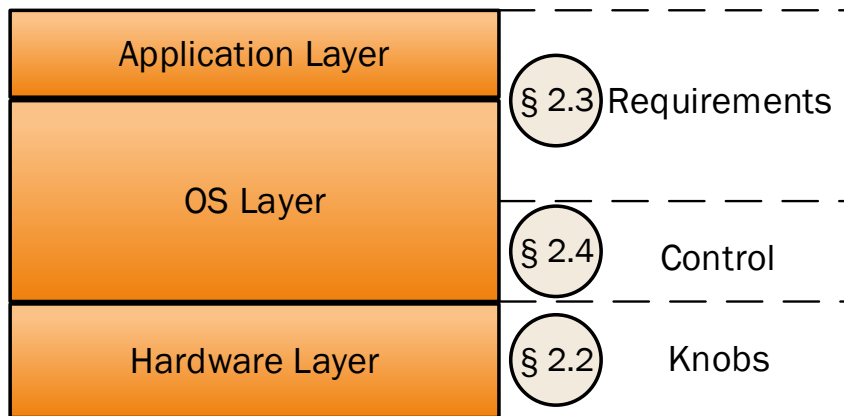


FIGURE 2.1: Flowchart of an embedded system in a cross-layer approach. Analytical approach (left) and practical approach (right). Sections 2.2, 2.3 and 2.4 are described as per the figure

Figure 2.1 presents the analytic division of an embedded system as a cross layer as defined before. Seen from a Power Management perspective, a Power Manager requires three things:

- The Power Knobs it will be manipulating. These knobs control the energy consumption of the processor, whether it is turning cores and modules on/off, reducing performance by reducing Voltage and Frequency (V-F), etc.
- The Control algorithm, which is the brain of the Power Manager. This algorithm takes decisions to modify the state of the Power Knobs. Depending on the application, normally the main objective of the Control algorithm is to reduce energy consumption of the processor with the least amount of performance penalties. This trade-off in energy and performance is regulated by the optimisation target.
- The Requirements provide the optimisation target to the Control. The Power Manager must aim to match the requirements which usually come in terms of a minimal performance requirement. In Real-Time Operating Systems (RTOSs) the application/task provides the performance requirement explicitly.

The elements required for the Power Manager can be mapped to the cross layer approach presented before, although the relationship is not necessarily one-to-one. The Control algorithm may be designed at any of the three layers, as an Application, as part of the OS or as a Hardware module. In the same way, the requirements may come from the Application running, or from the OS itself. This Chapter analyses all three elements of the Power Manager, with the Knobs available to the processor in Section 2.2, the Requirements and an overview of OSs in Section 2.3; and finally the Control algorithms for managing the knobs in Section 2.4, with a revision of the state-of-the-art of the algorithms available. Finally, the Discussion analyses the state-of-the-art approaches with their advantages and shortcomings. It addresses which improvements should be done to improve power optimisation.

2.1 Power consumption in Microprocessors

One of the major topics of concern when designing Integrated Circuits (IC) for battery-powered devices (WSNs) is the power consumption, as this will determine the active lifetime of the device before needing to replace or recharge the energy source. As technology has been developed, better energy storing cells have been designed, but this is not enough to fulfil the performance requirements. Therefore, some of the work is being done to reduce power consumption within the chip without compromising the performance constraint. The main objective of this project is to observe the power consumption of the OpenMSP430 and using low-power design techniques try to lower the power consumption overall. Therefore, the first step is to define power inside an IC and then the techniques that can be used to reduce its consumption.

The definition of *Power*, *Dynamic Power* and *Static Power* is based on [3]. The general definition of power is the energy consumed per unit of time. More specifically, the

energy E consumed is equal to the integral of the instantaneous power $P(t)$ over an interval T (Equation 2.1). The average power is then the Energy E over the interval T (Equation 2.2). In electric circuits, the instantaneous power is defined as the product of the instantaneous current $I(t)$ and the instantaneous voltage $V(t)$ (Equation 2.3).

$$E = \int_0^T P(t)dt \quad (2.1)$$

$$P_{avg} = \frac{E}{T} = \frac{1}{T} \int_0^T P(t)dt \quad (2.2)$$

$$P(t) = I(t)V(t) \quad (2.3)$$

For the architecture used in this project, the voltage is constant (set to 1.08V in the constraints). Therefore the instantaneous power $P_{total}(t)$ will be proportional to the sum of the currents of the transistors $I_{total}(t)$ while maintaining a constant V_{DD} .

$$P_{total}(t) = V_{DD} \cdot I_{total}(t) \quad (2.4)$$

The equation for the energy in the capacitor is given at Equation 2.5

$$E_C = \frac{1}{2}CV_C^2 \quad (2.5)$$

Given these equations, the total power consumed in a CMOS based IC is divided into the Dynamic Power and the Static Power (Equation 2.6). The former refers to the energy drained during switching of the CMOS gates, whereas Static Power is consumed regardless of the switching frequency.

$$P_{total} = P_{dynamic} + P_{static} \quad (2.6)$$

2.1.1 Dynamic Power

Ideal CMOS gates consume no power, as their output is connected to the input of another CMOS gate (the gate of a transistor, which is activated by voltage, not current). However, real CMOS gates contain a parasitic load capacitance at the output of the gate due to small capacitances intrinsic to the MOS transistors. Another “real world” peculiarity is that the input voltage switching is not instantaneous, so short circuit currents emerge due to this factor. Power dissipation coming from these non-ideal characteristics is called *Dynamic Power*. The two forms of Dynamic Power are (Equation 2.7):

- Switching Power
- Internal Power, dissipated by short-circuit currents (as defined by Keating et al. [2] and in the Synopsys Manuals [4])

$$P_{dynamic} = P_{switching} + P_{short\ circuit} \quad (2.7)$$

2.1.1.1 Switching Power

Switching Power results from the charging and discharging of the load capacitance C_{load} . Using Figure 2.2 as a reference of a CMOS inverter, when the input is 0 the PMOS transistor is ON¹, the capacitor C_{load} starts charging to V_{DD} (Charging in Figure 2.2). The output is 1. At this stage the NMOS transistor is OFF, so no current is flowing through it to V_{SS} . When the input is 1 (Discharging in Figure 2.2), the PMOS transistor is OFF (cutting V_{DD}), the NMOS transistor is ON and the capacitor C_{load} starts discharging through it. Therefore, using Equation 2.5 the energy of the load capacitance is then:

$$E_{C_{load}} = \frac{1}{2} C_{load} V_{DD}^2 \quad (2.8)$$

As V_{DD} is constant, the integral of the voltage yields V_{DD} . The total energy delivered from the Power Supply is then shown on Equation 2.9 [3]. Half of this energy is stored in the capacitor. The other half of the energy is dissipated as heat from the PMOS, because when charging the capacitor the transistor has a voltage across and it has a current flowing through itself [3].

$$E_C = \int_0^\infty I(t) V_{DD} dt = \int_0^\infty C \frac{dV}{dt} V_{DD} dt = C V_{DD} \int_0^{V_{DD}} dV = C V_{DD}^2 \quad (2.9)$$

Current is then driven through the PMOS only during the charging of C_{load} , so for this model, when C_{load} 's voltage is equal to V_{DD} the current is 0 (discarding leakage)

¹A transistor is considered ON when it is in its saturation region, in which a channel has been produced from Source to Drain. It is considered OFF when the voltage at the Gate V_G is the same as the Source V_S , so no path is created [3].

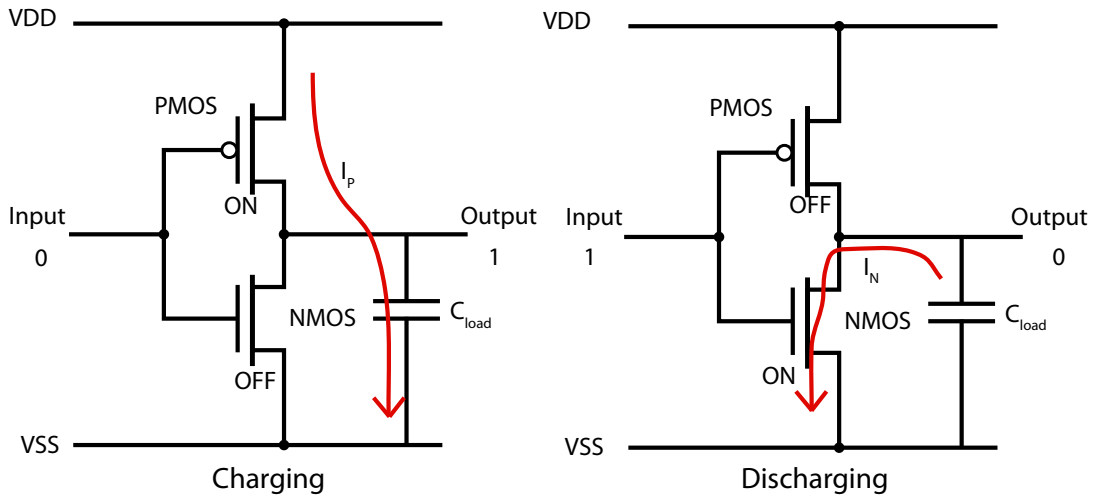


FIGURE 2.2: A CMOS Inverter showing the charging and discharging of the capacitor C_{load} . Based on [3]

power). In the same manner, after the capacitor has completely discharged following the activation of the NMOS transistor, the current flowing through the transistor is 0. Therefore, *Switching Power* is dissipated only during capacitor charge. To calculate the switching power in a CMOS circuit, the number of times it has switched have to be counted. Supposing the switching frequency is f_{sw} and T is the time over which the average Power is to be calculated, the number of switching times is $T \cdot f_{sw}$. Using this, (2.2) and (2.9), the switching power $P_{switching}$ is then [3]

$$P_{switching} = \frac{E}{T} = (T \cdot f_{sw}) \cdot \frac{CV_{DD}^2}{T} = f_{sw}CV_{DD}^2 \quad (2.10)$$

The previous calculation is for a CMOS gate that keeps switching every clock cycle. Real gate inputs do not toggle that often (except for clock inputs). Therefore, an activity factor α can be introduced based on switching probability for the device changing from 0 to 1 while the clock frequency f_{clk} is kept constant. The rearranged equation is then

$$P_{switching} = \alpha f_{clk} CV_{DD}^2 \quad (2.11)$$

2.1.1.2 Internal Power

Internal Power is dissipated when there is a short circuit current driven from V_{DD} to V_{SS} passing through both the PMOS and NMOS transistors. As mentioned before, non-ideal switching delays exist during transitions in the CMOS gates. During switching from “0” to “1” and vice-versa, there is a moment in which both transistors are in the linear region, which neither of the transistors are saturated nor shut down, so there is a direct path from V_{DD} to V_{SS} , effectively creating a short circuit. The name *Internal Power* is given based on [4] and [43]. Figure 2.3 shows the short circuit current. This current is

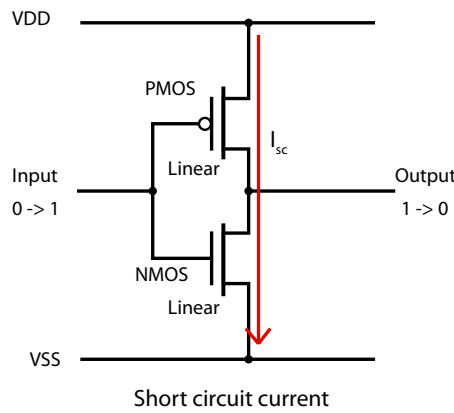


FIGURE 2.3: A CMOS Inverter showing the short circuit current flowing from V_{DD} to V_{SS} . Based on [4]

considerably smaller than Switching current because it only happens during short linear periods of both transistors. In this case, it is also obvious that power will be dissipated depending on the switching activity (αf_{clk}). Also, as the transistors are in the linear

(ohmic) region, they behave as voltage dependent resistors. The current $I(t)$ will depend on the voltage V_{DD} and the impedance dependent on V_{GS} .

As it can be seen, both types of Dynamic Power are dependent on the switching frequency of the gate and supply voltage V_{DD} . One approach could be to reduce V_{DD} (which in this case has been done, to a minimum of 1.08V), but this increases propagation delay. Therefore two options are possible, whether to reduce the general clock frequency, or the individual activity factor of the gates. The problem with reducing clock frequency is that although switching power is reduced, the energy required to complete a specific task will remain the same, as it will take the same number of clock cycles to be realised. Mathematically, using T_X as the time it takes to realise a particular task X , the switching count will be $T_X \cdot f_{clk}$. Using 2.2 and 2.11, the switching energy will then be

$$E_{switching} = T_X \cdot P_{switching} = T_X \cdot \alpha f_{clk} C V_{DD}^2 = (T_X \cdot f_{clk}) \cdot \alpha C V_{DD}^2 \quad (2.12)$$

$$\text{and} \quad T_X \cdot f_{clk} = k \quad \text{where } k \text{ is an adimensional constant} \quad (2.13)$$

$$\therefore E_{switching} = \alpha k C V_{DD}^2 \quad (2.14)$$

where, it is clearly seen that the Energy is not dependent on frequency.

2.1.2 Static Power

Static Power on the contrary does not depend on switching frequency but on the technology used. As the fabrication technology has been scaled down Static (or Leakage) Power has been increasing at the level that it equals and in some cases is greater than Dynamic Power [2], as shown on Figure 2.4(b). This means that even without switching, transistors dissipate energy only by being powered. Figure 2.4(a) shows three of the four leakage currents, which are (as stated on [2]):

Sub-threshold Leakage: Current flowing from drain to source when the transistor is in the weak inversion region. This is the most important source of leakage.

Gate Leakage: Current produced by the effect of tunnelling, where electrons flow through the Gate Oxide to the substrate, due to the reduced thickness of the oxide layer.

Gate Induced Drain Leakage: Current produced by the high field effect in the drain because of V_{DG} being high, which flows from drain to substrate.

Reverse Bias Junction Leakage: current produced by the creation of carrier pairs in the depletion regions.

As it can be seen from (2.15) (taken from [5]) (where K and n are constant, W is the width of the transistor, V_{th} is the threshold voltage, V_{DD} is the supply voltage and T

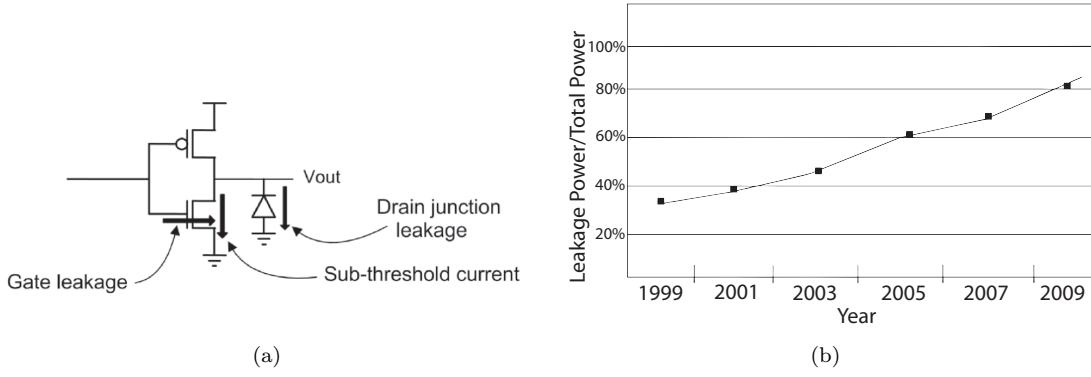


FIGURE 2.4: Leakage Currents in a CMOS circuit (a). Taken from [2]. ITRS trends for leakage power dissipation (b). Based on [5].

is the temperature), if V_{th} is reduced the subthreshold current increases exponentially. This happens with newer technologies, as the reduction in the technology size effectively reduces V_{th} [2]. As stated in the reference, current increases the temperature, and temperature increases I_{sub} exponentially, this problem is called *thermal runaway*. One way to reduce leakage power (as well as dynamic power) would be to reduce V_{DD} . As mentioned in section 2.1.1.2, reducing V_{DD} would increase propagation delay.

$$I_{sub} = KW e^{\frac{-V_{th}}{nT}} (1 - e^{\frac{-V_{DD}}{T}}) \quad (2.15)$$

Therefore, it is imperative that low-power design techniques are implemented in future processor designs to counter the increased power consumption due to Dynamic and Static Power.

2.2 Power Management Techniques: Knobs

2.2.1 Multi-Vdd

This technique is based on the premise that not all parts of the chip need to work at the same speed [44]. As mentioned in the introduction, the System-on-Chip (SoC) may contain several peripherals apart from the Central Processing Unit (CPU) cores that can run at different speeds. Therefore their supply voltage Vdd could be lowered and still meet timing requirements. Therefore its Dynamic (and Leakage) power could be lowered.

2.2.2 DVFS

Another widely used technique for reducing power consumption is DVFS, or also commonly found in the literature as Dynamic Voltage Scaling (DVS). This technique aims

to reduce power by reducing the voltage supply of the device (or the core) at the expense of reducing the frequency of the clock signal. This reduction of frequency is done because of gate switching latency. By reducing the voltage it can be seen in Section 2.1 that overall power consumption gets reduced. By reducing the clock frequency the device reduces its performance as computations are directly dependent on the operating frequency, so this technique is most effective when maximum performance is not needed.

Then, in order to use DVFS effectively, techniques have been designed to reduce power with the aim of penalising performance at the very least. These include the simple detection of what the workload is at a particular moment, to predict what the workload will be in the future. Other techniques are aimed to particular types of applications such as multimedia and videogames, where performance constraints are fixed in order to provide a desired quality.

Depending on the device, the DVFS implementation may be defined as either a discrete compact number of voltage-frequency pairs e.g. Intel's XScale PXA270 processor [45]; or as an adaptable range of frequencies with an adaptable range of voltages, as the one implemented by Nakai et al. [46]. This review focuses more on the former, as future implementations of algorithms following this report will be based in voltage-frequency pairs (from now on called *V-F pairs* or *V-F setting*).

2.2.3 PG

Dynamic Power Management (DPM) is a technique widely employed in microprocessors and other electronic devices, such as Hard Disk Drives (HDDs) and other peripherals. The process of DPM consists of taking advantage of idle times of the device (e.g. processor, HDD) between workloads, by setting the device to an available low power mode. Low power modes vary among devices, e.g. IBM's Travelstar HDD has 5 low power modes (plus active power modes)[6], the microprocessor StrongARM SA-1100 [6] has 3 power modes: **ACTIVE**, **IDLE** and **SLEEP**.

Normally there is an important trade-off between the power consumption (or power saving) in a particular power mode and the time and power it takes to transition to/from that power mode, meaning that the deeper sleep level i.e. the more power savings, the longer the transition time there is. Therefore the decision to go to a low power state mode should be based on the idle time available for the device before a workload arrives. In most cases (but not all) the power consumption for doing the transition to a low power state is higher than the power consumed when idle. A term is then defined as the **Break-Even Time** T_{BE} which is the amount of time the device should spend in the low power mode plus the transition time, for the energy consumption to be the same as the device in IDLE mode [6]. For simple explanation of the terms it will be assumed that a device has 3 power modes: **ACTIVE**, **IDLE** and **SLEEP**.

In microprocessors, the SLEEP mode reduces power consumption compared to just being IDLE, as several modules and clocks of the processor get disabled when sleeping. Ideally as soon as a workload is finished and the processor stops being ACTIVE, it would enter SLEEP mode. In reality, there is a transition period between ACTIVE/IDLE to SLEEP as well as from SLEEP to ACTIVE. Some processors have several sleep modes. For example, Texas Instruments' DM3730 processor [47] has 5 power states: Active, Inactive, Retention with logic on in low-voltage, Retention with logic off, and Off. Normally, the deeper the sleep level in a processor, the higher the overhead is, both in terms of time and consequently energy. These overheads are important to decide if it is worth going to a SLEEP mode. The **Break Even Time** is the minimum time the processor needs to be asleep for it to be worth going into sleep mode.

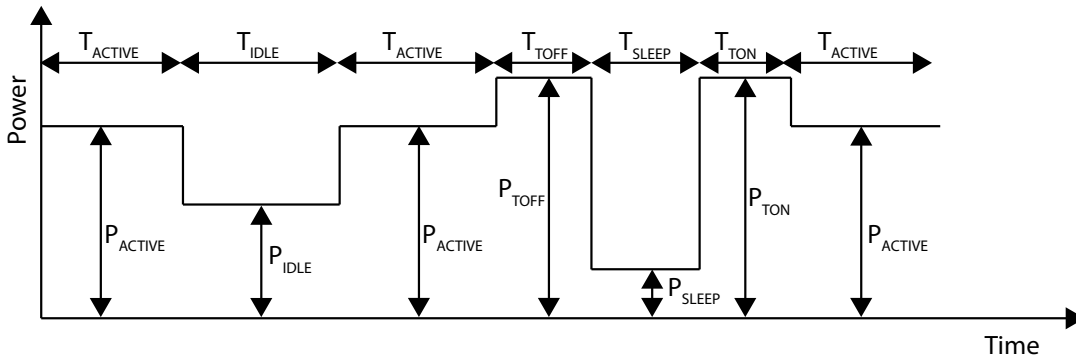


FIGURE 2.5: Definitions of power and times for a device with **IDLE** and **SLEEP** modes

In order to explain better what the Break Even Time T_{BE} is, Figure 2.5 shows the different powers and times under the different power modes and the transitions. Let P_{ACTIVE} be the power consumption when the processor is ACTIVE and the time at this state T_{ACTIVE} . In the same manner, P_{IDLE} is the power when the processor is IDLE, and the time at this state is T_{IDLE} . The energy consumed at ACTIVE and IDLE periods are E_{ACTIVE} and E_{IDLE} respectively:

$$E_{ACTIVE} = P_{ACTIVE}T_{ACTIVE} \quad E_{IDLE} = P_{IDLE}T_{IDLE} \quad (2.16)$$

For SLEEP mode there is a transition period T_{TR} , which is the sum of the time it takes to go to sleep T_{TOFF} and to wake up T_{TON} , which in turn these transitions dissipate power, P_{TOFF} and P_{TON} respectively. The sleep and wake up transitions are constant in terms of time and power, the energy it takes to transition E_{TR} is then fixed:

$$T_{TR} = T_{TOFF} + T_{TON} \quad E_{TR} = T_{TOFF}P_{TOFF} + T_{TON}P_{TON} \quad (2.17)$$

After the processor has gone to sleep, the power consumption P_{SLEEP} during the time T_{SLEEP} is the energy consumed E_{SLEEP} :

$$E_{SLEEP} = T_{SLEEP}P_{SLEEP} \quad (2.18)$$

Therefore, in order for the transition to the SLEEP mode to be worth it, it is necessary that the energy during sleeping E_{SLEEP} and the transition energy E_{TR} be less than the energy without going to sleep i.e. when being IDLE, this is E_{IDLE} :

$$E_{IDLE} \geq E_{TR} + E_{SLEEP} \quad (2.19)$$

The Break Even Time T_{BE} then substitutes the T_{IDLE} , so now T_{BE} needs to be calculated:

$$P_{IDLE}T_{BE} = \frac{E_{TR}}{T_{TR}}T_{TR} + P_{SLEEP}T_{SLEEP} \quad (2.20)$$

And as the T_{IDLE} is the time used for sleeping with the transitions, T_{SLEEP} becomes:

$$T_{IDLE} = T_{BE} = T_{TR} + T_{SLEEP} \quad \rightarrow \quad T_{SLEEP} = T_{BE} - T_{TR} \quad (2.21)$$

Substituting Equation 2.21 in Equation 2.20:

$$P_{IDLE}T_{BE} = \frac{E_{TR}}{T_{TR}}T_{TR} + P_{SLEEP}(T_{BE} - T_{TR}) \quad (2.22)$$

Solving for T_{BE} :

$$T_{BE} = T_{TR} \frac{\frac{E_{TR}}{T_{TR}} - P_{SLEEP}}{P_{IDLE} - P_{SLEEP}} \quad (2.23)$$

Based on this concept, the determining factor for deciding to go to **SLEEP** is then to know if $T_{IDLE} > T_{BE}$. As in a microprocessor this value is not known *a priori*, different techniques have been developed to estimate what T_{IDLE} will be before really knowing it.

Benini et al. [6] provides a thorough review of different implementations of DPM techniques and their advantages. To provide some context for more complex techniques explained later, some of the basic DPM techniques of the reference are explained here. DPM techniques for microprocessors are mainly divided into Predictive and Stochastic Techniques.

2.3 Applications and OS: Requirements

Nowadays, high performance embedded systems are able to run applications that demand plenty of resources, intensive processing power, dedicated accelerators, internet for streaming, memory for running simultaneous applications, etc. It is a responsibility of the system not only to run these applications, but to provide a decent performance for each of them whilst doing it, with a finite power supply (battery) powering the system.

The simplest strategy to save energy would be to reduce V-F using DVFS. The problem is the reduction in performance, which in turn impacts user experience. Therefore, in order to reduce the energy consumption whilst preserving performance, a Power Manager is required to control the processor's power states. As opposed to setting a V-F manually, a *dynamic* Power Manager can control the V-F on the run, based on the OS, application and user needs. These needs or *requirements* allow the Power Manager to adapt and compensate for the situation.

The requirements needed from a Power Manager can come from a particular application or from the OS. There are two types of OSs, General Purpose Operating System (GPOS) and RTOS. In GPOS e.g. Windows, Linux, iOS, these requirements come from the OS, where the main optimisation constraint is the idle time, sampled at regular intervals. This approach allows to save energy, but as it is oblivious of the task or application being run, the Power Manager is not concerned in the user perceived performance. As opposed to GPOS, the main objective of RTOS is to provide a required performance to given tasks. In order to provide a performance that reduces energy consumption but does not compromise user experience, it is possible to take some elements from RTOSs for a Power Manager. Section 2.3.1 provides the definition of RTOS, an explanation as well as the types of existing RTOSs.

2.3.1 Real-Time Systems

The main objective of a Real-Time System (RTS) is to execute tasks within a time given. The system's tasks must not only deliver a result but deliver it *timely*[48]. The two main aspects of RTSs are to aim to meet the deadlines, and to behave in a predictable manner. In order to cope with several applications that require real-time scheduling, complex RTSs run in an RTOS.

The most important concepts in a Real-Time System are Deadlines, Latency, Jitter, Predictability, Worst Case, and Periodic Task. Taken from the book by Sally [10], they are described:

Deadline The time at which a task must be finished. This is the most important component of the real-time system, it is what makes it real-time. In hard real-time systems failing to meet the deadline can result in disastrous consequences.

In soft real-time systems the value of the task decreases when completed after the deadline.

Latency The time between what should happen and when it does. Ideally latency should be zero, but in reality, hardware and software components take time (if infinitesimal) to move electric signals from one place to another. For example, a hardware interrupt caught in a pin of a processor activates a software interrupt. Even though this is almost instantaneous, there is a time taken between these two events i.e. latency.

Jitter It is the variance of latency. In reality, latency between events varies in time. A non-deterministic task scheduler in the OS (e.g. Linux) cannot ensure that a task will start running at an exact given time with a given latency.

Predictability Is the ability to know how much time will a task take to complete. A completely predictable system can repeat a process with no variation in the results nor in terms of the time taken. In reality, in order to make a real-time system predictable, jitter and latency are included when calculating the execution time (worst case).

Worst Case Latency and jitter are factors in real-time systems. In order to make the system predictable, when determining the execution time of a task, a worst case scenario is calculated, which includes latency and jitter. Therefore, the task scheduler will know what is the longest time the task will take, in order to decide how to schedule it to meet its deadline.

Periodic Task A task that executes at periodic (regular) intervals, and the interval of the task is always the same.

Priority Inversion This happens when a task with low-priority holds a lock on a resource needed from a high-priority task, keeping the high-priority task from executing.

2.3.2 Hard Real-Time versus Soft Real-Time

The two main types of RTSs are *Hard* and *Soft* RTSs. The former require all of the deadlines be met, if not, serious consequences may happen e.g. a lagging heart rate monitor. The Soft RTS on the other hand are more relaxed and can tolerate deadline misses. At a deadline miss, the value of the result at the end of the task is reduced, but it does not result in heavy consequences. Examples of this are media players, as a loss in performance translates into reduced frame rate. Table 2.1 shows a comparison between Hard and Soft Real-Time.

As mentioned, Hard and Soft RTSs are aimed at different situations. Observing a general purpose system, in order to increase power savings whilst maintaining performance, it

	Hard Real-Time	Soft Real-Time
Worst case performance	Same as average performance	Some multiple of average performance
Missed deadline	Work not completed by the deadline has no value	Output of the system after the deadline is less valuable
Consequences	Adverse real-world consequences: mechanical damage, loss of life or limbs	Reduction in system quality that is tolerable or recoverable, for example lagging video
Typical use cases	Avionics, medical devices, transportation control, safety-critical industrial control	Networking, telecommunications, entertainment

TABLE 2.1: Hard Real Time versus Soft Real Time comparison, as explained by Sally [10]

may be feasible to take elements from a Soft Real-Time perspective. The complexity of implementation may arise due to the fact that the scheduler in a general purpose system is not optimised for real-time. Also, GPOS Power Managers are usually in the OS layer, and they have no interaction with the applications.

2.4 Power Management: Control

2.4.1 DPM

2.4.1.1 Predictive Techniques

These techniques base their decisions of going to **SLEEP** mode on the observation of previous idle times to predict what the near future idle time will be. The terms *underprediction/overprediction* are defined as the situations in which the predicted idle time is shorter/longer than the real idle time respectively.

Under Static Predictive Techniques one of the simplest algorithms is the *Fixed Timeout*. In this case the assumption is that under idle time, after a T_{TO} the device will be idle for a long time. For this algorithm, when a workload is finished a timer is started. When it reaches T_{TO} the device is put in **SLEEP** mode. The efficiency of the algorithm is then dependent on the selection of T_{TO} .

Some other Static Techniques include *Predictive Shutdown* (mentioned by Benini et al. [6]) and *Predictive Wakeup* [49], in which one of the techniques is to compute a non-linear regression equation based on history to predict future idle times. The problem with *Fixed Timeout* and *Predictive Shutdown* is the performance penalty paid as both

algorithms base their decisions on whether or not the device should **SLEEP**, not really on when the device should wake up.

Abbasian et al. [50] shows a prediction method using Wavelet Forecasting in an Event-Driven environment. This algorithm uses the Wavelet Transform, and intends to predict what the idle time will be between workloads to decide if the device should go to sleep or not. Similar to the previous citation this is a *Predictive Shutdown* method which in any case that the device goes to sleep will incur in a loss of performance.

The *Predictive Wakeup* [49] attacks the problem of performance penalty, as it is done by predicting what the idle time is going to be, so that the device should be in the **SLEEP** mode for a particular time and then wakeup. In the cases of *underprediction* power will be wasted but this will provide less performance penalties overall.

These Static Techniques by themselves may not produce good results, as they depend on offline information as well as the device being stationary to be somewhat effective. They can be improved by being Adaptive, so that the control parameters can be modified on the run. Some techniques include Adaptive Timeouts in which the T_{TO} is modified based on recent history, as mentioned by Benini et al. [6].

The implementation of Hwang and Wu [49] is complemented as an adaptive algorithm called Exponential Weighted Moving Average (EWMA), with the prediction of T_{IDLE} as a weighted average of the past history. It is a recursive algorithm, where the prediction is saved as the new average each time new data arrives. The average is multiplied times a parameter (λ) less than one, which then exhibits an exponential decay. Let $w(n)$ be the last real idle time period, \bar{w} the last predicted idle time period, $W(n)$ the predicted idle period, and λ the attenuation factor. Equation 2.24 shows how the predicted idle period is calculated. After finishing the workload, if $W(n) > T_{BE}$ the device goes to **SLEEP** for the predicted time.

$$W(n) = \lambda \cdot w(n) + (1 - \lambda) \cdot \bar{w} \quad (2.24)$$

The factor λ determines how important is the recent and old history. A high λ gives more significance to recent events, whereas with a low λ recent history has little effect. Upgrades made to this algorithm are to include a timeout policy for *underprediction* cases e.g. waking up a lot earlier than needed. The drawback with the Hwang and Wu [49] approach is that it does not perform well in a high frequency change of idle times due to its absolute dependence on history.

2.4.1.2 Stochastic Techniques

Predictive Techniques assume that workload arrival is deterministic. Stochastic Control on the other hand models the system as an optimisation problem with uncertainty [6]. Policy optimisation intends to map states with decisions to make based on a stochastic model. This allows to have a more complex system with different power states (other than **ACTIVE**, **IDLE** and **SLEEP**). Implementations of these techniques include a modelling of the device or system as the Service Provider (SP) which processes workload requests issued by the Service Requester (SR).

Benini et al. [6] present an approach to DPM using Markov chains and Markov Decision Processes (MDPs). The service requests, or the workload arrivals to the processor are modelled as a Markov chain, called the Service Requester (SR). A Markov chain is seen as a stochastic process formed by states, in which transitions from one state to another are defined by a probability. These transitions follow the Markov property, which states that the transition to a next state only depends on the current state, and not on the history of the previous states. This makes the system *memoryless*, as the previous states do not influence on the probability for the next state. The SR is represented in Figure 2.6 a), where the two possible states R are 0 being no new requests, and 1 being a new request to the provider. The Service Provider (SP) is then the processor. The authors of the paper used a StrongARM SA-1100 which has one sleep mode. This SP is modelled as a MDP, which is an extension of a Markov chain, with the difference that this system has Actions and Rewards. The states S are *on* and *off*. The DPM power manager is in charge of taking the actions A , being switching *on* to *off* with s_{off} and *off* to *on* with s_{on} , as well as preserving the current state. This is represented in Figure 2.6 b). The function for taking the actions is then $f : S \times R \rightarrow A$, which is the basis for the policy required to optimise, with the objective of reducing power consumption at the minimum performance penalty.

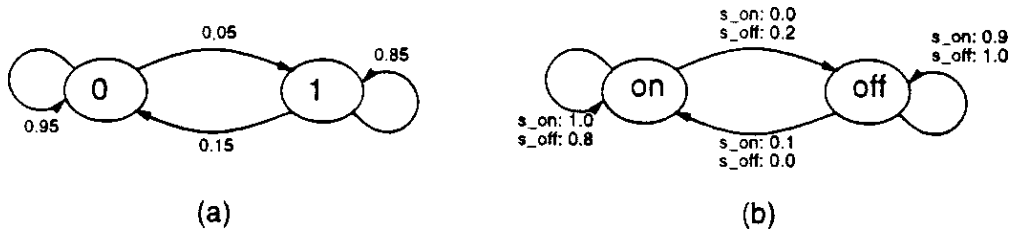


FIGURE 2.6: Markov chain and Markov Decision Process model for DPM of StrongARM SA-1100, as presented by Benini et al. [6]

For this type of model time is divided in discrete periods in which the probability of having a request on period $n + 1$ depends on the state at period n . The *PM* policy is designed to take decisions based on both the state sets and the action set. As mentioned in [6], the Markov model has to be known *a priori*, techniques suggested include Linear

Programming for optimisation of the model. Some important aspects to notice of Markov models are that there is no guarantee that the expected values for power and performance are going to be optimum. If the model is not accurately designed, the policy will give approximate optimal solutions.

That approach is also limited by being applied to stationary workloads, which are unlikely to occur in real life. An adaptive approach reviewed by Benini et al. [6] overcomes the stationary limitation by having a policies for different workloads. In this scheme, the Power Manager observes the environment (workload requests) and selects the most suitable policy for that workload profile. As pointed out by Shih and Wang [21], this approach adapts right to the workload in turn, but the drawback is the amount of memory and computations required.

Other stochastic techniques used in the literature using Markov Chains include the Continuous Time Markov Decision Process *CTMDP* and Time-Indexed semi-Markov Decision Process *TISMDP* both mentioned by Simunic et al. [19].

2.4.1.3 Machine Learning Techniques

Dhiman and Rosing [51] present a design of a *PM* that includes a Machine Learning algorithm for selecting among a range of DPM policies. This algorithm, instead of presenting a new DPM technique, learns online which technique is suited best for the current performance constraint. The DPM techniques in this implementation are called *experts*, and for this particular paper the experts used are: *Fixed Timeout*, *Adaptive Timeout*, *Exponential Predictive* and *TISMDP*, all explained before.

The algorithm is based on updating a weight vector \mathbf{w} composed of N experts in which each w_i (with $i = 1, 2, 3 \dots N$) represents the weight of an expert. The weight of each expert represents the performance of said expert, in which a higher weight represents higher performance. At the start all weights are set to sum to one, and from there each expert may increase or decrease its value depending on its performance.

When an idle period starts an expert is selected using the one with the highest probability from the probability vector \mathbf{r} which is calculated as:

$$\mathbf{r} = \frac{\mathbf{w}}{\sum_{i=1}^N w_i} \quad (2.25)$$

Each expert's probability is then $0 \leq r_i \leq 1$ and the sum of all r_i 's is 1. The selected expert is called *operational expert* and all others are *dormant experts*. After the selection, the device is controlled by the operational expert. When the idle period ends all experts are evaluated, the operational expert with its performance and dormant experts as how they would have performed if they were selected. This speeds up the learning process.

The evaluation of the experts is done taking in account both performance and energy savings. The loss vector \mathbf{l} holds all the loss factors l_i in which each loss is calculated in Equation 2.26. l_{ie} represents the loss of energy and l_{ip} represents the loss of performance, and the factor α ($0 \leq \alpha \leq 1$) is the performance constraint, or the relative importance of energy savings vs. performance.

$$l_i = \alpha l_{ie} + (1 - \alpha) l_{ip} \quad (2.26)$$

In this implementation energy loss l_{ie} depends on whether the device went to **SLEEP** mode, so if $T_{idle} < T_{BE}$ (defined at the beginning of Section 2.4.1) there was no **SLEEP** mode so $l_{ie} = 1$. If the device went to **SLEEP** mode, l_{ie} is calculated as Equation 2.27, where T_{sleepi} is the sleep time of expert i greater than T_{BE} .

$$l_{ie} = 1 - \frac{T_{sleepi}}{T_{idle}} \quad (2.27)$$

For performance loss l_{ip} there is no implementation of predictive wakeup so the device will wakeup when a new request comes. Therefore if the device went to **SLEEP** mode $l_{ip} = 1$ and if not $l_{ip} = 0$. At the end of the algorithm the update of the weight factors is done as Equation 2.28 where β is a constant between 0 and 1. This means that a higher loss l_i decreases the weight and a low loss l_i increases it. Algorithm 1 shows the flow of the DPM implementation.

$$w_i \leftarrow w_i \beta^{l_i} \quad (2.28)$$

Algorithm 1 Algorithm for expert based machine learning. Taken from [51]

- 1: $\beta \in [0, 1]$
 - 2: $\mathbf{w} \in [0, 1]^N$ such that $\sum_{i=1}^N w_i = 1$
 - 3: **loop**(for each T_{idle}):
 - 4: Choose expert with highest probability factor in $\mathbf{r} = \frac{\mathbf{w}}{\sum_{i=1}^N w_i}$
 - 5: Idle period starts \rightarrow operational expert performs DPM
 - 6: Idle period ends \rightarrow evaluate performance of experts
 - 7: Set new weight vector \mathbf{w} to be: $w_i \leftarrow w_i \beta^{l_i}$
 - 8: **end loop**
-

This paper holds that the selection and evaluation of experts is done during the **ACTIVE** period so no overhead is incurred when reaching the **IDLE** period. It is stated that the performance of the algorithm will be that of the best performing expert on the expert set, so care is taken when selecting the range of experts. As stated before the performance constraint is controlled by the α factor, in which a higher α will save more energy at the cost of greater performance penalties, whereas a low α will improve performance with a higher energy consumption.

This algorithm was also implemented within DVFS policies as well as DPM-DVFS policies together, and they are mentioned as part of the DVFS Techniques on Section 2.4.2 and the Interplay of DVFS and DPM on Section 2.4.3 respectively, both by Dhiman.

2.4.1.4 Reinforcement Learning for DPM

Reinforcement Learning is a widely studied field of Machine Learning which has gained popularity because of its simplicity and adaptiveness to the environment. This method is based on the learning done in nature, as the process is done on a trial and error basis under an unknown and possibly ever-changing environment, accumulating experience based on past history. The implementation of Run-time Manager (RTM) using Q-Learning is explained in Chapter 3 which is loosely derived from a Reinforcement Learning method called Q-Learning. A brief description of Reinforcement Learning and Q-Learning is given in Section 2.4.4.

Some implementations of Reinforcement Learning have been done in the field of DPM and DVFS [20, 24, 25, 29]. The reasons for this are the adaptiveness of the system under changing conditions of the environment, so it may adapt to non-stationary situations; as well as the simplicity of the implementation.

Tan et al. [20] show a DPM technique that finds the best policy of going to **SLEEP** states based on experience. The system environment is modelled to be a SR, a Service Queue (SQ) and a SP, in which Reinforcement Learning (RL) states are given by all three modules. Actions are executed by the Power Manager *PM* on the SP in the form transitions to power modes **SLEEP** and **ACTIVE**, and the reward function (in this case cost function) is modified to not only take in account for energy savings but also performance delay, using a Lagrangian multiplier λ as the trade-off parameter of energy-performance.

The system may not be Markovian but the RL algorithm is still implemented, at the cost of slower convergence. To speed up convergence the PM is designed to not only evaluate the visited state but the possible states that could have been visited by the SP, had the actions been different, something similar as with [51]. Another improvement for faster convergence is the variable learning rate α that is decreased with the pass of time.

The reference [25] continues the implementation of [20], in which the cost function energy-performance trade-off λ is tuned on-line based on a performance constraint. This performance constraint is fixed in the paper to be a latency of the device of 5%, in which a latency lower than this number is said to have converged. If the latency increases again (to over 10%), the λ parameter is further tuned.

Another RL implementation is done in [24], in which a hardware architecture acts as the PM. In this case an algorithm named *SARSA* (State-Action-Reward-State-Action),

similar to Q-Learning; is implemented. This approach is similar to the one in [51] in the sense that the decisions taken are to select which DPM technique is going to be used on the next idle time, similar to the *experts* of [51]. The policies (experts) implemented are *Always On*, *Greedy*, *Timeout* and *Stochastic*. Depending on the workload the PM adjusts itself to use the policy that suits best. The size and power consumption of the PM is not documented so it is not possible to compare it to the implementation done in this report.

2.4.1.5 Other Predictive Techniques

Another technique developed in the literature for DPM is shown on [21] with an algorithm called Adaptive Hybrid Dynamic Power Management *AH-DPM*, which predicts both the duration idle time in both bursty and non-bursty periods. This technique adapts the timeout explained in Section 2.4.1.1. The *PM* determines then if the device should be shut down and what its timeout period should be. Bursty and non-bursty periods are monitored as history doing exponential average. Results on this paper show that power savings are greater and performance penalties are smaller than other algorithms analysed such as stochastic [6], static and adaptive timeouts [6], predictive [49] and machine learning [51].

A different approach was taken by Gniady et al. [26] and Hwang et al. [23] in which prediction is done by looking at the Program Counter (PC) of the processor to identify device calls to particular peripherals. This is done by monitoring and analysing the patterns of access to I/O devices at runtime. So by looking at PC patterns the *PM* may be able to predict when the I/O devices will be accessed, in this way the device is woken up in advance so that performance penalties are minimum.

Following the predictive algorithm of Hwang and Wu [49] about exponential average idle time prediction, explained in Section 2.4.1.1; some adjustments were done by Chen et al. [28]. This work addresses the issue when the distribution of idle periods changes sharply. As the conventional exponential average algorithm gives more importance to recent events (idle periods) if the next event changes drastically the algorithm will not respond accordingly. Therefore the proposal is to enhance the Equation 2.24. Same as in [49], the idle time prediction is only used to determine whether $I_{n+1} > T_{BE}$ so that the device goes to **SLEEP** mode. As there is no predictive wakeup every time the device goes to **SLEEP** there will be a performance penalty when returning to **ACTIVE**.

Remembering the notation of Hwang and Wu [49], let i_n be the last real idle time period, I_n the last predicted idle time period, I_{n+1} the predicted idle period, and a the attenuation factor. The enhanced Equation 2.29 shows the b factor. This is included to optimise the algorithm by adjusting the predicted values to the changed distributions. Equation 2.30 shows how the b factor is calculated, and as it is stated by Chen et al.

[28], if the ratio of the two last idle periods i_n and i_{n-1} is between the interval of U_{min} and U_{max} , the change is mild so no changes are done in the prediction. If the ratio is outside the interval of U_{min} and U_{max} , there was a large change between the 2 past idle periods, so the ratio is included as part of the prediction.

$$I_{n+1} = b(a \cdot i_n + (1 - a) \cdot I_n) \quad (2.29)$$

$$b = \begin{cases} 1 & \text{if } \frac{i_n}{i_{n-1}} \in [U_{min}, U_{max}] \\ \frac{i_n}{i_{n-1}} & \text{if } \frac{i_n}{i_{n-1}} \notin [U_{min}, U_{max}] \end{cases} \quad \text{where } U_{min} < 1 < U_{max} \quad (2.30)$$

The algorithm performs notably better than the presented by Hwang and Wu [49] although it does not address the performance penalty at wakeup.

It is important to note that these techniques explained are implemented mostly on HDD's [20, 21, 23, 26, 51] and Network Cards (WLAN NIC's) which have a behaviour that may not represent a processor's in terms of workload input or transition times, but the principle used for the implementation of adaptive, predictive and stochastic techniques is key to be applied on different devices.

2.4.2 DVFS

2.4.2.1 Workload Detection

To select an appropriate DVFS setting without penalising performance it is fundamental to know what the workload is, regardless of how it is represented. The more information there is describing the workload, the better decisions can be taken on the V-F setting. Dhiman and Rosing [45] divides DVFS techniques in three areas. 1) Some proposals assume that the deadlines, task arrival times and workload are known in advance. 2) Other techniques need some sort of compiler support in which sections of the program/-code/task can be characterised offline so when section x of code is reached a particular V-F setting is selected. 3) No information from the software level is known in advance, so the PM needs to adapt to the incoming workloads and model the task behaviour. Added to these techniques, an important area of development includes the implementation of machine learning to not only adapt but predict what the workloads and deadlines will be in the future.

2.4.2.2 Online Learning

Dhiman and Rosing [45] uses the same algorithm as the one proposed for DPM in Section 2.4.1.3 but this time for DVFS. For this case, the V-F setting is selected based on the workload of the processor. Performance Counters are used to determine what the workload is, in which a high workload means that the instructions are more focused on CPU operations; and a low workload is oriented more on memory access operations, so the processor could run at a lower frequency. The XScale PXA270 processor is used in this paper, which allows to obtain 4 different performance counters simultaneously. Based on these performance counters, the variable μ is calculated, and it represents the workload percentage ($0 \leq \mu \leq 1$), where $\mu \rightarrow 0$ means more memory-oriented instructions and $\mu \rightarrow 1$ means more CPU intensive instructions.

Each V-F setting represents an expert (recalling Section 2.4.1.3), and it is estimated that each expert bound to a particular μ range, so a higher μ would require the device to be at a higher frequency. The algorithm adapts by penalising experts that incur in both performance and energy losses, updating the weight and probability vector to select the expert that may perform better at the given workload. The same as the implementation by Dhiman and Rosing [51], the factor α is defined by the user, which gives more importance to either energy saving or performance.

One of the most important contributions of this paper is that it quantises workload as a single percentage number by combining different performance counters, which can help simplify task characterisations in future work.

An important note by Dhiman and Rosing [45] is that the performance and energy losses shown in the results are relative to the workload and selection of the expert, and may not map directly to real performance and energy losses, so an energy loss affecting the cost function of the implementation does not represent real energy lost but an estimation and assumption of the energy model.

2.4.2.3 Offline Learning

Another approach from Moeng and Melhem [7] shows offline training and characterisation of workload. V-F settings tuned up offline, in which performance metrics are monitored, and goal metrics which in this case are the energy per (user-instruction)² ($epui^2$) are measured. Workload is characterised and mapped, so a particular combination of metrics is mapped to a V-F setting. Performance metrics are monitored at runtime. For a particular combination of performance metrics, the V-F setting with the lowest $epui^2$ is selected.

Several *performance counters* can be found in the processor and are available for analysis. From these counters the performance metrics are derived. In the work of Moeng and

Melhem [7] the different performance counters are analysed (and combinations of them) to find a correlation between them and the $epui^2$, in which the three most significant ones are selected as performance metrics.

Machine Learning is used to create a Decision Tree because having a permutation of all metrics would be very large and impractical at run-time. The Decision Tree is created by extracting features that can represent the whole range of possible metrics in just a few nodes (*leaves* of the tree). In the end the result is a lookup table (*LUT*) in which the addresses of the LUT are generated by combinations of the metrics, so a particular combination of the metrics has a V-F setting at which $epui^2$ is the smallest. Figure 2.7 shows an example of a Decision Tree in which 3 metrics (as selected by Moeng and Melhem [7]) are mapped to a particular frequency (which would correspond to a V-F setting).

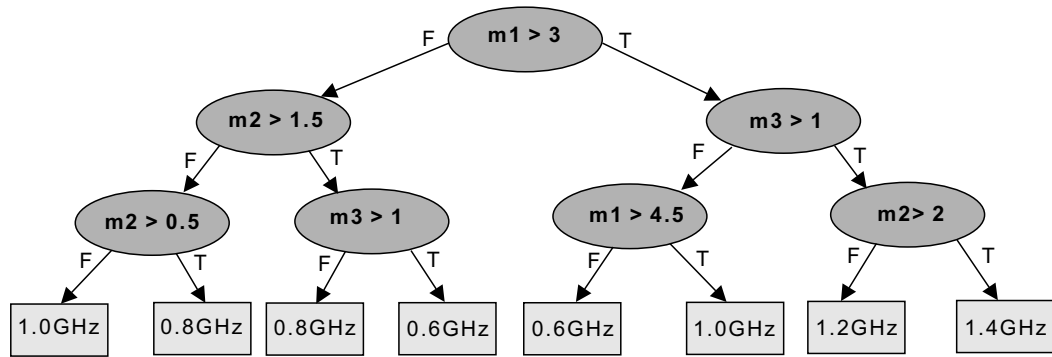


FIGURE 2.7: Decision Tree example for selecting V-F settings. Taken from [7].

This paper’s approach makes the “on the run” computation relatively simple, and as the implementation was done in hardware, the overhead is not significant. The downside is that the efficiency for savings and performance depends on the workloads used for the offline training, so if a particular type of workload was not present at the time, or if the granularity of the mapping of metrics to V-F setting was not optimal, the solution will not be optimal. It is mentioned in the reference that for unknown workloads the implementation should work thanks to the metrics classification, but there is no guarantee that this will happen.

Another important contribution from this paper is that the design is aimed towards multicore processors, and by doing the training offline the V-F selection is done simple regardless of the number of cores.

AbouGhazaleh et al. [52] presents a supervised learning approach to generate a DVFS policy based on workload. This implementation not only sets the frequency of the processor (CPU) but of the cache memory as well. A special-purpose compiler (called *PACSL*) is proposed which runs sample applications and generates a policy to map workload states to V-F settings. This workload is characterised by 5 parameters, so in a similar approach as Moeng and Melhem [7], decisions of a new DVFS setting are taken

when certain conditions present in the parameters. Also similar to Moeng and Melhem [7], the learning is carried in two phases

1. The training to obtain the performance counters' measurements and identify what they call "states" of the system (combination of different performance counters).
2. Develop rules (like the Decision Tree [7]) to map the states to V-F settings, so that the Energy-Delay product is the least.

2.4.2.4 Workload Prediction

In [53–55] they implement a "neuromorphic controller" that predicts the workload percentage. The algorithm was implemented in Hardware and the function performed by the neuromorphic controller is an exponential weighted moving average. It is unclear how the reference quantises performance loss, delay, or performance hit. How do they know the selected V-F setting is the appropriate one for the particular workload. The performance loss mentioned in the paper is due to a misprediction of the workload, taking in account the transition time from one V-F setting to another, so in the case of a misprediction the change of V-F setting would have to be double (one to the predicted workload and one to get to the real workload).

Sinha and Chandrakasan [56] proposed to have a DVFS (DVS in the paper) predict workloads using different adaptive filters. Workload is characterised and predicted by the update of the coefficients of a series of adaptive filters. The ones implemented are *Moving Average Workload*, *Exponential Weighted Averaging* (EWMA), *Least Mean Squares* and *Expected Workload State*. It is mentioned in the paper that the filter with the lowest prediction RMS error is the Least Means Square using 3 taps (past workload states).

2.4.2.5 Deadline Prediction

One of the most important ways of quantising performance loss is to compare execution time of a particular load to a timing reference. If the execution time is larger than the timing reference, or *deadline*, it can be said that there was a performance loss. This deadline can be either assigned explicitly by the OS for the cases of RTOS as explained in Section 2.3.1, or created synthetically by using information of the system, such as the frame rate in a video player, or the arrival of a request for a new workload.

There are cases in the literature where deadlines are created synthetically the system can be treated as a soft RTOS, in which a term is defined as an "effective deadline" [57, 58]. For applications without an explicit deadline, it is required that the workload is done in an acceptable amount of time. When a new workload arrives the system assumes

that the previous workload has already been finished. If this is not the case, workload requests accumulate and what is seen is a performance loss. Therefore to ensure this accumulation does not happen, the effective deadline is set to be the arrival of a next workload. So by knowing this effective deadline the PM can select the lowest V-F setting that still reaches the deadline. As sometimes the arrival of the workload is not known in advance, the effective deadline is unknown. Some algorithms in the literature intend to predict this arrival to know what the remaining time is for processing the workload.

2.4.2.6 Application-specific DVFS

Multimedia applications present an attractive field for Power Management for two reasons: workloads have very similar characteristics throughout the whole multimedia processing; and second, video/audio/games require a to have a particular frame rate. A frame rate lower than the one expected will result in a performance loss noticeable for the user.

Ge and Qiu [29] presents an interesting approach to manage Multimedia Applications. In this case instead of Power Management the focus is on what is called Dynamic Thermal Management, in which temperature throughout the chip is monitored and fed to the Thermal Manager for it to make decisions. The Q-Learning algorithm (Section 2.4.4) is implemented for this paper, in which the states of Q-Learning are given by current workload plus the temperature of the chip, and the possible actions to take are the V-F settings.

The paper by Gu and Chakraborty [36] shows a Control-based approach aimed at videogames. They implemented a PID (Proportional-Integral-Derivative) controller which at each frame of the videogame it decides which V-F setting will be best for the predicted workload. As the frame rate is a major performance constraint, if workload is predicted accurately it is possible to know exactly which V-F setting will work best without a performance loss. The implementation of the PID computation cost is negligible compared to the time per frame. The coefficients of the PID are fixed but there are some tunable parameters for the controller.

Also in the area of application-specific DVFS, the work of Choi et al. [34] has focused on prediction techniques for energy optimisation. They proposed an energy saving technique aimed towards MPEG video decoding using DVFS. The technique is based on three factors:

- Each MPEG video frame is divided into Frame-Independent (FI) and Frame-Dependent (FD) workloads. The FI part is constant for each frame type, whereas the FD part may be variable.

- The FD part is predicted using the EWMA algorithm. After the FD workload has been predicted, the V-F setting for that frame is selected arbitrarily.
- The use of intra-frame compensation: errors in the prediction of the FD workload are compensated with the FI part, so if the FD was “over-predicted” and its workload was finished earlier, the V-F setting for FI is decreased to save energy whilst still meeting the deadline. On the contrary, if FD was “under-predicted”, the V-F setting for the FI part is increased to still meet the deadline.

This was an interesting approach for an application-specific approach. The EWMA results provide low prediction errors as the prediction was done per frame type. It is important to mention for this approach that being applied to a real world situation, it may be necessary to analyse the DVFS costs in terms of timing overheads and energy consumption. Also, doing DVFS changes means a change in the context, from user level to kernel level, which generates an extra timing overhead. The video decoding was run at 2 frames-per-second (fps), so it may not be realistic in terms of timing.

Bang et al. [35] present another application-specific DVFS algorithm, a follow up to Choi et al. [34]. Their approach does workload prediction as well, with workload prediction per frame type. The prediction algorithm uses Kalman Filters. This is an interesting approach because uncertainty is included as part of the prediction using Kalman Filters. The results show higher accuracy predictions than control algorithms (PID controllers), but energy savings are not significant.

2.4.2.7 General Purpose DVFS

Current popular OSs have power management systems to increase battery life. Some of the techniques are static in the way that they reduce power consumption at the expense of compromising performance. Examples of this are Windows ‘Power Saver’ mode. In Linux the Power Managers are called Governors. These Governors are Linux Kernel Modules[59–61] which operate at Kernel space. The static governors are set to fixed V-F. For example, the ‘powersave’ governor is fixed to the lowest V-F available, where as the ‘performance’ governor is set to the maximum V-F. The ‘userspace’ governor allows for any available frequency to be selected. Dynamic governors on the other hand control the V-F settings at run-time, varying them to adjust for the load. The most common dynamic governors are ‘ondemand’ and ‘conservative’[17]. The principle behind these governors is to reduce energy consumption within a performance constraint. The constraint is to have a minimum idle time percentage. The governor checks the workload percentage at periodical intervals. If the minimum idle time threshold is surpassed it means that a high workload has arrived, so the governor changes the V-F setting accordingly. After the change, the governor starts decreasing the V-F until it reaches

the minimum idle time again. This is a clever method to reduce energy consumption, although the two main drawbacks are:

- The ondemand and conservative algorithms are not predictive but reactive, so they assume that the next interval will have the same workload as the current interval. Therefore, the governors also depend on how frequent is the workload percentage sampling: seldom sampling and the system may waste too much energy (running at high frequencies) or it may be providing low performance; too frequent sampling and the system may change V-F too often, possibly increasing the DVFS overhead.
- The governors are oblivious of the applications running, and the only constraint is the idle time measured, so even though energy may be saved, this does not mean that the application is outputting the desired user experience.

2.4.3 The Interplay of DPM and DVFS

It is argued that both DPM and DVFS techniques can work together complementing each other to obtain the maximum power reduction with a performance constraint [22, 62].

Dhiman and Rosing [22] proposes to merge the DPM of Dhiman and Rosing [51] and the DVFS Dhiman and Rosing [45] online learning algorithms to have an interplay of both. Experts for both DPM and DVFS are implemented and selected based on the weight vector mentioned in Section 2.4.1.3 and Section 2.4.2.2.

Their experiment shows that for *Idle-Dominated Workloads* the DPM policies show to be more effective, whereas in *Computationally Intensive Workloads* DVFS expert selection proves to be the best choice. Therefore a much larger range of workloads can be handled. As with the separate implementations of DPM and DVFS, the performance is constrained to the α factor, in which a higher α represents a higher saving but higher delay, and a low α produces higher performance at the cost of less energy savings.

Bhatti et al. [62] shows a similar approach (in fact it is based on Dhiman and Rosing [22]) with a weight vector, but aimed to Real-Time Systems. Their implementation called *HyPowMan* shows a very similar algorithm to the one by Dhiman and Rosing [22], with the main difference that the experts are aimed towards Real-Time Systems, so the DVFS experts are scheduled with Deadline oriented policies. Tasks are profiled with 4 elements called release time, Worst - Case Execution Time (WCET) (Worst Case Execution Time), relative deadline, and periodicity of the task. These elements help the algorithm decide which combination of DPM and DVFS experts to select.

2.4.4 Reinforcement Learning

Reinforcement Learning (RL) sits in a spectrum between Supervised and Unsupervised Learning, in which feedback is somewhat informative. The principle of RL is the ability for an agent (controller) to make decisions that affect its environment and from that to obtain useful information and feedback so that it can learn to make better decisions in the future. It is inspired in human and animal learning [8], in which one learns from one's mistakes. The simple diagram of RL is shown on Figure 2.8. Some works on RL are done by Busoniu et al. [8], Sutton and Barto [63], Sutton [64], Ribeiro [65], Szepesvári [66].

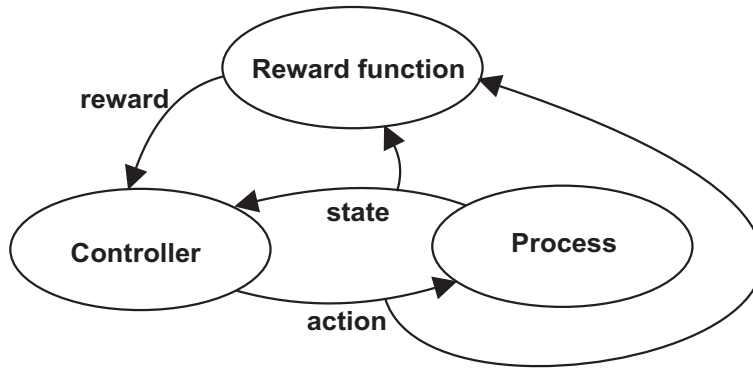


FIGURE 2.8: Reinforcement Learning simple diagram. Taken from [8]

As RL is centered on making decisions, it is all about control of a process in an environment by finding the optimal behaviour. The elements in RL are:

- The Agent
- State Space S
- Action Space A
- A reward function $R : S \times A \rightarrow R$

The objective of RL methods is to maximise the long-term accumulated reward. Therefore a policy π must be learned based on experience, which maps the states to actions. By states it is meant that the environment is in a particular shape or with particular identifiable properties. The actions represent the possible decisions that could be taken when in a particular state. The reward function evaluates how good or bad was the action taken by the agent.

RL is designed to be implemented in a model-free environment, so it must adapt without previous knowledge of it. In order to reduce complexity for the algorithms, the state space is designed to be discrete, finite and relatively small, although there are cases where function approximation is needed as the state space is very large (or infinite).

Symbol	Description
s and s'	Current State and Next State
a and a'	Current Action and Next Action
$Q(s, a)$	Q-Value at State s and Action a
α	Learning Rate
r	Reward
γ	Discount Factor

TABLE 2.2: Symbols of Q-Learning algorithm

Another approach (actually used in the implementation on this report) is to map a large state space into a small set of discrete states.

One of the most popular algorithms for RL is *Q-Learning*, which aims to find the optimal policy based on previous rewards. The algorithm for Q-Learning is seen on Algorithm 2. The symbols used are shown in Table 2.2. An important requirement for the convergence of the Q-Learning algorithm is to have the **Markov Property**, which states that the state at $t + 1$ does not depend on the past states $t, t - 1, t - 2, t - 3 \dots$ but only on the immediate state (and action) at t . This memoryless property allows the algorithm to be compact. In any case, Q-Learning may be implemented in systems that are not strictly Markov, but assumed to be.

The Q-Learning algorithm works by observing the environment (obtaining s), takes an action a and reaches another state. The new state gives a positive or negative reward depending on whether the action taken was good or bad. The “learning memory” is kept in a *Q-Table* which contains the *Q-Values* of each state-action pair. The Q-Value $Q^\pi(s, a)$ represents the long-term expected reward if the policy π is followed when action a is taken in state s .

Algorithm 2 Q-Learning algorithm. Taken from [63]

```

1: initialise  $Q(s, a)$  arbitrarily
2: loop(for each episode):
3:   initialise  $s$ 
4:   repeat(for each step of episode):
5:     choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.  $\epsilon$ -greedy)
6:     take action  $a$ , observe  $r, s'$ 
7:      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
8:      $s \leftarrow s'$ 
9:   until  $s$  is terminal
10: end loop

```

At the start the Q-Table should be initialised. If previous knowledge of the system exists (in the form of an already filled Q-Table), the working Q-Table can be initialised to the known values. Otherwise it is initialised to an arbitrary value. At every episode (as named in Algorithm 2) the state s is identified and an action a is taken with the policy constructed from the Q-Table. For taking the action there are several options to

be followed. A simple solution is called ϵ -greedy, shown on Algorithm 3. As there is no previous knowledge of the system in the beginning, the decisions should be exploratory. When the agent is confident of its knowledge of the system, the decisions can be taken towards exploiting the action a with the highest Q-Value.

After the action has been taken, the environment is observed to obtain the reward r which shows how good or bad the action was taken. The Q-Value of the action a taken at state s is updated using the equation in Algorithm 2 line 7. In this equation, the learning rate α ($0 \leq \alpha \leq 1$) determines how the discounted reward affects the current value of $Q(s, a)$. The discount factor γ ($0 \leq \gamma \leq 1$) introduces a pseudo-horizon for the policy to follow.

Convergence of the algorithm is stated at Busoniu et al. [8], Sutton and Barto [63], proven that the iteration of the values occurs an infinite number of times.

Algorithm 3 ϵ -greedy strategy. Based on [8]

```

1: function CHOOSE_ $a$ ( $\epsilon, s$ )
2:   generate random  $x$ 
3:   if  $x < \epsilon$  then                                     ▷ Exploitation
4:      $a \leftarrow \arg \max_a Q(s, a)$ 
5:   else                                                  ▷ Exploration
6:      $a \leftarrow$  random action
7:   end if
8:   if  $\epsilon < 1$  then
9:     increment  $\epsilon$ 
10:  end if
11:  return  $a, \epsilon$ 
12: end function

```

2.5 Discussion

High performance embedded systems have been presented, where one of the main concerns for these systems is energy consumption. Having plenty of power management areas available, such as peripheral, memory, and processor power management, the latter has been selected as the topic of study. For this, the cross-layer approach was introduced, where it was proposed that the power managers are composed of three aspects, low power techniques available, requirements for performance, and intelligent control of the low power techniques based on the requirements and observations of the environment.

RTSs were presented, which provide a requirement to the system for finishing the tasks in a given time. The Soft Real-Time (RT) behaviour presents an attractive paradigm to be used in modern devices running GPOSs. Nowadays there are implicit performance expectations from a device, and the Soft RT approach provides a quantifiable method to deliver that performance. Energy adjustments can be done knowing that a deadline needs to be met. Therefore, the requirements for energy optimisation and performance could be provided by the application to the Power Manager, and it could adopt a Soft RT behaviour.

An exhaustive review of the state-of-the-art power management algorithms was done. In terms of DPM using PG, the approaches presented include the use of predictive techniques such as EWMA, stochastic techniques using Markov chains, machine learning with expert based learning, and Reinforcement Learning. The main concern observed with PG and DPM has been that in reality it may be not feasible to offer real-time performance using DPM. For long periods of idle time DPM may be the best solution.

Using DVFS for power management has also been presented. The six types of DVFS control techniques analysed are:

Offline Learning The advantage of this technique is the task profiling using performance counters. The main drawback of this approach is the fact that it is offline profiling, so the algorithm does not adapt to new situations. Also, it is oblivious of the applications running, as the profiles are based solely on the performance counter metrics.

Online Learning The main advantage is the adaptiveness to the workloads arriving. The main drawback as its DPM counterpart is the reliance on the energy-performance trade-off parameter, which does not ensure real-time performance. There was a combined DVFS - DPM version of the expert-based algorithm, with the same advantages and drawbacks.

Workload Prediction These algorithms bring an optimisation question, on how frequent the workload sampling is optimal. Similar to online learning, the energy savings were determined by the energy-performance parameter.

Deadline Prediction The approaches taken here treat the system as Soft RT, where the algorithm aims to provide real-time performance by creating “effective deadlines”. Even though the aim was to offer real-time performance, the algorithm did not know of the application, therefore the performance for the applications could not be quantised.

Application-specific DVFS These approaches have provided substantial energy savings with acceptable performance. They are application-specific, so there is no flexibility for applying the algorithm to different applications.

General Purpose DVFS The Linux Governors were analysed. Their main limitation is as mentioned before, where there is no contact with the applications, so there is no knowledge of the application performance.

After observing the different proposals, it is clear that there are limitations in the current control algorithms. There is then the need to provide a new approach for Power Management. One of the main limitations seen is the lack of communication with the applications, so a cross-layer approach is key to create this synergy between application, OS and hardware. Another key element observed is that in order to provide decent user experience, it is necessary for a Power Manager to quantise its performance against the application. This can be achieved by the aforementioned Soft RT approach. Given that offline profiling may not be feasible for multiple applications, it may be necessary to create an application model at runtime, so prediction and decision algorithms are required. Prediction to know what kind of workload is going to be processed *a priori*, and decision to know which V-F to take, even with the presence of noise and uncertainty. An overview of Reinforcement Learning was presented as an alternative to learning to make decisions at runtime. Finally, the main limitation of the application-specific algorithms can be fixed by enabling the runtime manager to work with different applications, allowing them to provide their requirements to the manager. This may be done by creating a common programming framework where developers may be able to write their applications to be power-aware.

Chapter 3 then shows the theoretical proposal for a runtime manager to cope with the limitations of current approaches. Chapter 4 presents the implementation of said runtime manager, and Chapter 5 lists the results obtained from experimenting with the proposed design.

Chapter 3

System-Level Power Management

3.1 Run-Time Management

Computing systems nowadays can be represented as a system of three layers, namely the *Applications*, the *Operating System (OS)* and the *Hardware*. The purpose of the system is to provide functionality to the user by means of running the applications e.g. document writing, sending e-mail, watching a video. The OS is responsible for managing runtime of the applications by scheduling them, allocating running time for each of them when having simultaneous applications, allowing resources to them, etc. It is also responsible for managing the hardware resources for processing the applications, for communication, and control of the hardware peripherals. This includes the control of the power management hardware available, enabling/disabling the different power levels and sleep states.

The control algorithms for power management can reside at any of the three layers. The power management algorithms can be implemented as hardware modules that trigger the power states[24, 53, 54]. The OSs provide interfaces to add extra functionality to them and control the power states at the OS level e.g. the power governors in Linux[17, 18]. The algorithms are also implemented as small applications that can request power state changes to the OS, which reduces the complexity of implementation[27, 34, 35].

Independent of the implementation on the layers, the power management algorithm is called *Run-time Manager (RTM)*. The RTM is then responsible of making energy efficient decisions, based on the constraints it has, whether the priority is to increase performance, power consumption, reduce temperature, etc. The constraint will be called *Requirement* (Section 3.1.1). The decisions are executed by using the low power techniques, namely Power Gating (PG) and Dynamic Voltage and Frequency Scaling (DVFS). These techniques are called *Power Knobs* (Section 3.1.2). The decisions for using the Power Knobs are defined as *Control* (Section 3.1.3). In order to make efficient energy decisions it is necessary to observe the environment to measure the performance yielded,

the energy consumed, the battery life, or the temperature increase. These observations are called the *Monitor Feedback* (Section 3.1.3.1). The different elements of the RTM need to communicate for the power management to work. Therefore the RTM uses a cross-layer integration. A graphical description of the cross-layer RTM is shown on Figure 3.1¹ as an example generic diagram. The Requirements are sent to the RTM. The Power Knobs control the power states of the Central Processing Unit (CPU) by order of the RTM sending Voltage and Frequency settings. The Monitor Feedback is provided by the Performance Monitors.

The work presented addresses the need for a cross-layered RTM to optimise energy consumption for an application aiming to achieve given performance requirements by it (the application). Therefore, the applications targeted by the RTM have to have a particular quality, which is an inherent performance requirement. The Power Knob studied in this research is DVFS. Given that DVFS controls the speed at which the workload clock cycles are executed, the performance requirement from the application must come in the form of a timing requirement. The next three subsections explain the three main components of the RTM: Requirements (Section 3.1.1), Power Knobs (Section 3.1.2) and Control (Section 3.1.3).

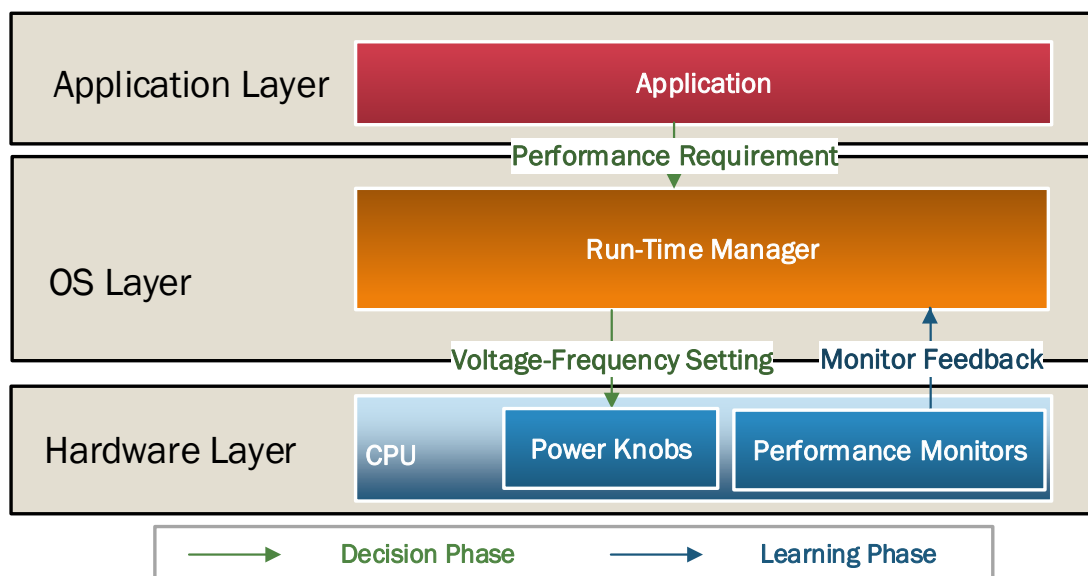


FIGURE 3.1: Generic cross-layer RTM, where the arrows show the communication between the layers.

3.1.1 Requirements

² The aim of this research is to provide higher energy efficiency to platforms running General Purpose Operating Systems (GPOSs). Hard Real-Time Operating Systems

¹ Modified the Figure to have a clearer explanation

² Added this new subsection

(RTOSs), as explained in Section 2.3 which exhibit deterministic timing behaviour and have hard deadlines, are out of the scope of this work. The RTMs is then designed in a cross-layered GPOS environment, analysing the non-deterministic timing nature of these OSs. Examples of this environment are Linux, Windows and Android. The aim of this work is to relate GPOSs to have a Soft Real-Time (RT) behaviour. In GPOSs the applications are not performance constrained explicitly as the ones in an RTOS, therefore no effort is done by the OS scheduler to achieve a specific performance for the applications.

Applications running in an RTOS provide a performance requirement in the form of deadlines for the workloads to be executed, therefore allowing the RTM to obtain energy savings by reducing the performance of the workload whilst achieving the deadline. The RTM proposed is aimed to sit in the GPOS spectrum but to provide an RTOS-like behaviour. This means that the target applications must have a non-necessarily explicit performance requirement for it to be the optimisation target. A non-necessarily explicit performance requirement means that the scheduler is not notified of the requirement. The timing requirement can be seen as the maximum time allowed for the workload to be finished. The unit of workload is then defined as a frame. Therefore, the performance requirement is then defined as the number of workload frames that need to be executed per second, or frames-per-second.

The other major component of an Real-Time System (RTS) is the Worst - Case Execution Time (WCET) which allows the scheduler to decide how to run the task. In a Hard RTOS the WCET must be provided, whereas in GPOSs this is not given. This means that in a GPOS the WCET of the workload to be processed needs to be calculated. The control algorithm is then responsible for this calculation, given that the application can provide annotations to help this calculation be more precise.

The environment for the RTM is then a system in which the workload for an application is divided in frames, and this application has an implicit requirement for the frames to be executed in a specific time, defined as a deadline. These frames can be of different workload sizes i.e. different amounts of CPU Cycles per frame. An example of an application working in this environment is video decoding. In video decoding, the video frames represent the workload. The implicit performance requirement is the frames-per-second (fps) at which the video must run, which are usually contained in the video file itself. The deadline can be calculated as the reciprocal of fps i.e. $deadline = \frac{1}{fps}$, as the time allowed for each frame. Also, in this environment resources are limited, including memory. It is assumed that frames are processed at the fps speed i.e. if the frame is finished processing before the deadline, the system is idle until the deadline is reached, when a new frame is brought to be processed. Therefore, idle time cannot be accumulated. The reason behind it is that in the systems targeted there is not enough memory to use as a processing buffer, so only one frame can be processed and buffered at the time.

In this environment, the power management knob available is DVFS, so no PG is taken under consideration as part of the RTM. The race-to-idle[32] approach, in which the strategy is to process the workload as fast as possible and reduce energy consumption during the idle time cannot be applied to this environment.

The cross-layer RTM allows applications to send their requirements (or constraints) to it, which then adjusts its behaviour to achieve those constraints. The constraints analysed in this research are time based, meaning that the RTM must achieve the highest energy savings whilst providing a performance measured in units per second. As the RTM works in a non-deterministic timing environment, it is impossible to ensure Hard RT behaviour, as some deadlines could be missed. The objective is then to achieve Soft RT performance. The main difference between Hard and Soft RT performance is the criticality of not achieving the desired performance when executing the tasks. Soft RT is addressed then, because the task workload is not known *a priori*. Therefore, without offline profiling of the tasks, the task completion time cannot be known before execution. The RTM must then attempt to finish the workloads before their deadlines by collecting information at run time, but some may be missed.

Efficient RTMs need requirements in order to have a target to aim for, whether it is to reduce energy consumption, control temperature or provide an amount of performance. The goal of this work is to provide energy savings whilst achieving Soft RT performance on GPOSs. Soft RT behaviour means completing tasks before reaching their given deadlines, and where missing a deadline is not critical but seen as a performance penalty. These performance penalties (or lack thereof) have an impact in user experience.

The intention is then to target applications that have an implicit minimum performance requirement. It is implicit because the requirement is not given to the OS and therefore is not considered when scheduling the application. A distinction is done into *general purpose applications* and *Soft RT applications*. The former do not have implicit requirements, so lower performance of these applications does not have a significant impact on user experience, and include text processors, email clients, etc. Soft RT applications have an implicit constraint, whether this is frame rate or throughput (bits per second). Examples include video processing, rendering, video games, music players, video chat, etc. Watching a video with a lower frame rate than intended can have a negative impact on user experience.

For the RTM to provide Soft RT behaviour, the implicit performance requirements need to be passed to it. These performance requirements are also called annotations, so the application requires extra code to include annotations for it. The RTM needs these implicit requirements and annotations to become explicit to have them as the performance targets.

3.1.1.1 Application adaptation for use with RTM

³ The RTM requires the target applications to provide annotations to it in order to have an optimisation target, as well as to provide information regarding the structure of the application. The former represents the performance requirement in fps. The latter means that the RTM needs to be notified when a new frame of the application is to be executed. The application then requires to be divided into frames. In order to improve energy optimisation even further, the new frame notification should include a type in order to classify frames. This is further explained in Section 3.2.

For the sake of example a video decoder is presented as the target application. The video decoder opens a video file, which contains the fps embedded in the video header. The application code needs to be modified to send the fps requirement to the RTM as soon as it has been read from the file. The RTM now has its optimisation target. The video decoder starts playing the video, decoding each frame to be displayed. The video decoder code needs to be modified to send the frame start notification to the RTM as soon as it is reached in the application. The RTM notification from the application includes the frame type. After all frames have been decoded, the application finishes execution. It sends a notification to the RTM of the end of the application.

In order to simplify the adaptation of the application code to include the annotations, an Application Programming Interface (API) was designed. This API consists of C functions that are inserted inside the code at design-time. Section 4.5 provides a description of the implementation of the API framework.

3.1.2 Power Knobs

The RTM for this work uses DVFS as its Power Knob. The use of processors with long sleep transitions added to the unpredictable behaviour of the scheduler in GPOS can make using PG impractical. As there are several processes being managed by the scheduler, added to the unpredictability of system interrupts, sleep transitions and the energy overhead associated to them may prevent PG from providing actual savings. PG is currently not included as part of the RTM.

As mentioned before, DVFS implemented in System-on-Chips (SoCs) use a discrete number of Voltage and Frequency (V-F) pairs. The RTM designed needs then to learn the list of V-F pairs available to make the proper selection.

³ Added this new subsection

3.1.3 Control

Control is the main decision element on the RTM where the algorithm executes. The RTM then The Control algorithm can be placed in any of the three layers of the system (Hardware, OS or Application). Chapter 4 refers to where the RTM has been implemented. As mentioned before, in order to provide Soft RT performance the algorithm needs the constraints given by the Requirements. The algorithm decides based on its learning experience what the most suitable power state is by adjusting the Power Knobs.

3.1.3.1 Monitor Feedback

A key element needed for the RTM to learn is the Monitor Feedback. It provides the performance data to adjust the algorithm in real time. The data comes from the Performance Counters (or Performance Monitors), which are directly connected to the processor and its components. The data is read from the counters upon request.

3.1.4 Run-Time Management Algorithm

The RTM has then a power minimisation objective for the applications, which translates into the solution to a constrained optimisation problem. The effective solution to the problem has two requirements:

1. The workload needs to be known before being processed so that it can be performed at the lowest V-F setting.
2. The decision taken once the workload is known (or estimated) needs to fulfill the time-based constraint achieving the deadline for the current frame, taking into account the application's variations.

The RTM was then designed to address these requirements. The algorithm of the RTM was named **Shepherd**. The first objective is achieved by predicting the workload of the frame and the second objective by learning how to make the V-F decisions based on the predicted workload of the frame. Therefore, the original generic design presented in Figure 3.1 is then modified to incorporate the needs of the Shepherd algorithm. The new RTM is shown in Figure 3.2. To facilitate the design of the RTM algorithm, it was split into two modules (or units), these are the **Prediction Unit** and the **Decision Unit**, respectively. This simplifies the problem, so that the algorithms inside each unit can be modified, upgraded and tested in isolation. The Prediction Unit is explained in Section 3.2. The Decision Unit is seen in Section 3.3. The steps followed by Shepherd are explained in Algorithm 4.

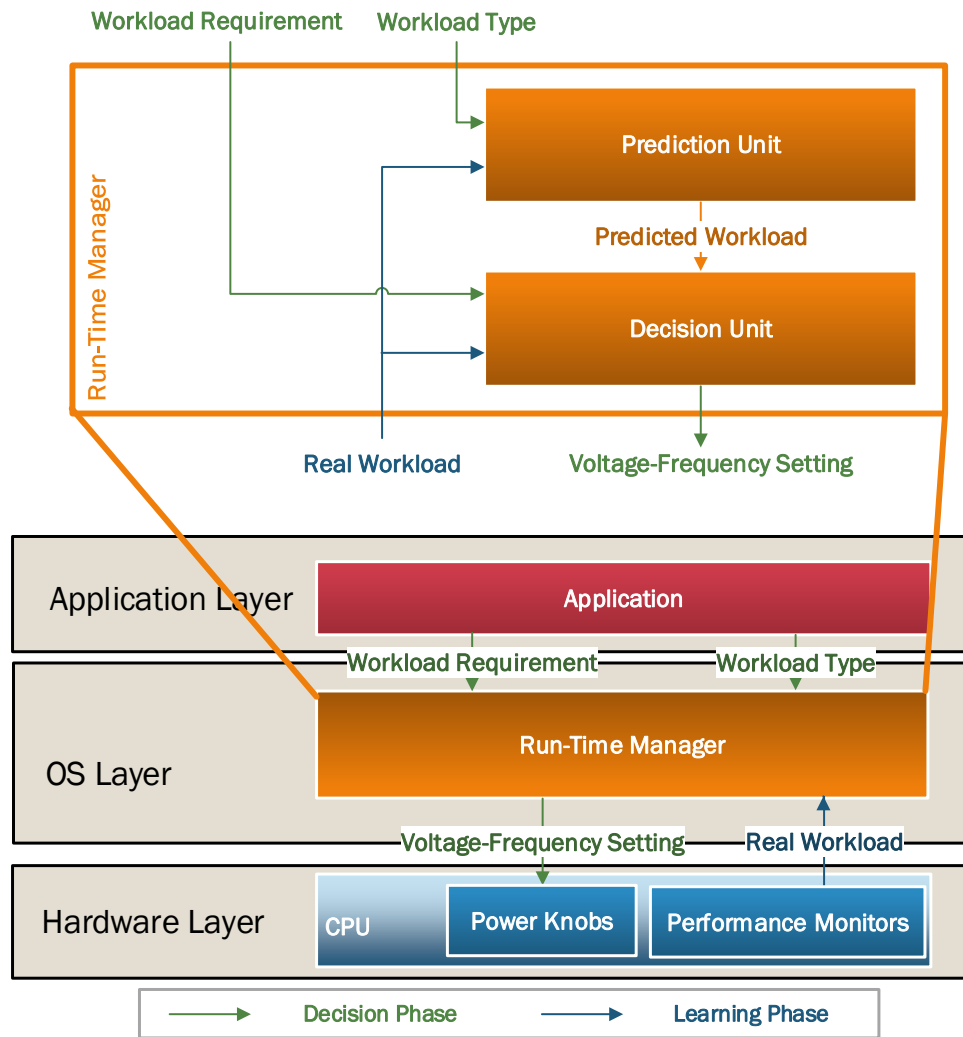


FIGURE 3.2: Run-Time Management Unit in the cross-layer approach

The RTM runs together with the application and it is triggered by it. When the application starts the RTM is dormant waiting for a request. When this request comes it provides the requirements to the RTM. The application then runs. At the start of a new frame Shepherd predicts the workload for that particular frame, based on this prediction it then decides the suitable V-F pair to achieve the deadline, and then the frame is executed. After finishing the frame, the RTM collects performance data from the performance monitors and then it ‘learns’ by updating the Prediction and Decision Units.

Algorithm 4 Shepherd Power Management

```

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

```

3.2 Prediction Unit

3.2.1 Motivation

The scenario for generic Real-Time Systems includes knowing the Worst - Case Execution Time for each task to be run. In General Purpose Operating Systems this is not possible to be known a priori, so this gives the RTM the need to estimate it. The Prediction Unit is in charge of providing an acceptable estimation of the workload for the next frame.

A possible technique could be offline profiling of the application prior to running it, obtaining the workload to be run at each frame, creating a complete workload profile of the application. The implications of this technique are that they depend on profiling each application before running, added to the memory required to store these profile values. Also, the complexity of the profiling is increased in a realistic setting. This because of the overheads of the scheduler, OS and other background processes running concurrently in the same system. After profiling the workload of an application, reproducing the conditions in which the workloads were obtained may present a problem on its own.

Therefore, an estimation or predictions of the upcoming workload must be obtained. There is a range of prediction algorithms that can provide a decent estimation of the workload. From this range, one of the main constraints for selecting the algorithm is the time it requires to process, or the timing overhead; as the algorithm itself should not impact the performance of the processor significantly. Another aspect of the algorithm is the precision of the predictions. Given that the SoCs studied exhibit a discrete and relatively small number of V-F pairs, the precision of the prediction is less important. This because the difference of the performance yielded by two power states is marginally different. For example, the DM3730 [11] SoC clock frequencies are $300MHz$, $600MHz$, $800MHz$ and $1GHz$, so the selection of frequency is done based on the more significant figures of the predicted performance.

3.2.2 EWMA

The Exponential Weighted Moving Average (EWMA) algorithm was selected to perform workload prediction. The EWMA algorithm is widely used in the literature [34, 53, 56] for workload prediction because of its lightweight implementation and acceptable performance. 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:

$$W(n) = w(n) \cdot \lambda + \bar{w} \cdot (1 - \lambda) \text{ where } 0 \leq \lambda \leq 1 \quad (3.1)$$

where $w(n)$ is the measured workload at time instance n measured from the hardware, \bar{w} is the average workload in the time interval 0 to n , $W(n)$ 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:

$$\bar{w} = W(n) \quad (3.2)$$

A 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 type⁴ e.g., for type 1, $W_1(n)$ 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, 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. Figure 3.3 shows the timeline of a sample video (frame 430 to 615) where the real workload running a video (red) and the predicted workload (green) can be seen. The Workload is measured in CPU cycles. Each point in the X axis is a frame to be decoded and its point in the Y axis shows the amount of CPU cycles taken to decode that frame. The green line shows the predicted CPU cycles for each frame.

⁴For a video processing, workload type translates to the frame type i.e., I, P and B frames.

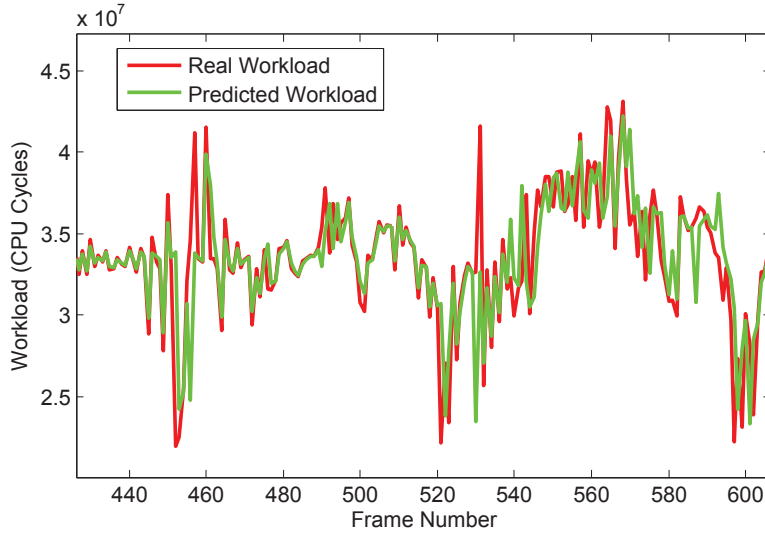


FIGURE 3.3: Comparison of real workload vs. predicted on a sample video, from frames 430 to 615

3.2.2.1 Optimisation and Results

As shown on the literature[35], variations in the workload of an application are dependent on the workload “type” i.e. for video processing, frames of the same type present low variations. Therefore, for the Governor to have more precise predictions, it requires the frame type (workload type) as one of its parameters every new epoch. In order to demonstrate the need to group the frame types furthermore, an experiment has been carried out to check for variations in the workload. The workload was measured for 3 videos of h.264 VGA resolution (640x480) run at 24 fps, for 30 seconds, run 100 times each. Each video has 3 frame types ⁵ (I, P and B) and has a total of 715 frames. On average of the 3 videos there are 22 I frames, 411 P frames and 282 B frames. The analysis done has been obtaining the standard deviation of the workloads of each video to illustrate the variation of the workloads. On one side the standard deviation was calculated for all the workload frames of each video. On the other side the workload frames were grouped by frame type, and the standard deviation was calculated per group. The average was calculated for each standard deviation of the 3 videos and 100 runs. Table 3.1 shows the comparison of the workload frames grouped and ungrouped. Frame grouping by type yields lower variations in the workload, particularly on I type frames. Splitting the workload into frame types can then take advantage of this, so that the EWMA can be calculated per frame type.

The implementation of the EWMA filter does include only one parameter λ , which determines the importance of previous data. In order to use the EWMA filter, the optimal

⁵For sake of uniformity, frame type names are mapped from video, i.e. I, P, B, to 1, 2 and 3 respectively

		Workload standard deviation (CPU Cycles)
No grouping		2.23×10^7
Frame types	Type 1	1.68×10^6
	Type 2	1.47×10^7
	Type 3	1.93×10^7

TABLE 3.1: Comparison of variation of the workload with and without grouping into frame types

parameters were obtained by analysing different workloads, as shown on Figure 3.4. Figure 3.4 shows the optimisation of the parameter λ run with different videos. The videos represent Dynamic Workloads, as each frame presents variations in its workload. It can be seen that beyond a value of 30% for λ , the Mean Absolute Percentage Error (MAPE) is reduced below 4%. The duration of the videos was of 720 frames (30 seconds for a 24 fps video). The same analysis was carried using a Static Workload (FFT) (Figure 3.5), this means that the application was running the same workload, in this case for 100 frames. As there are no variations in the workload, the prediction error MAPE goes around 1%. It is safe to say that a value above 30% for λ gives a good prediction.

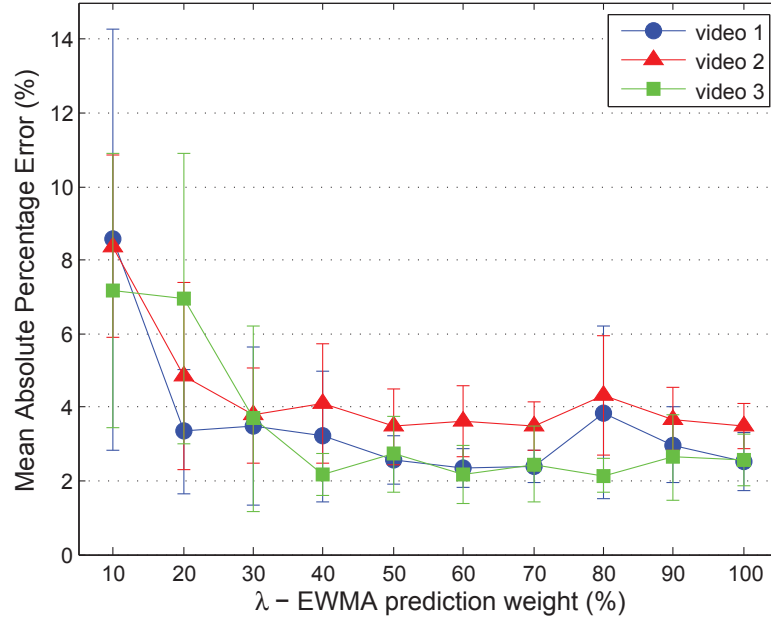


FIGURE 3.4: Effect of weight for EWMA on prediction error using dynamic workloads

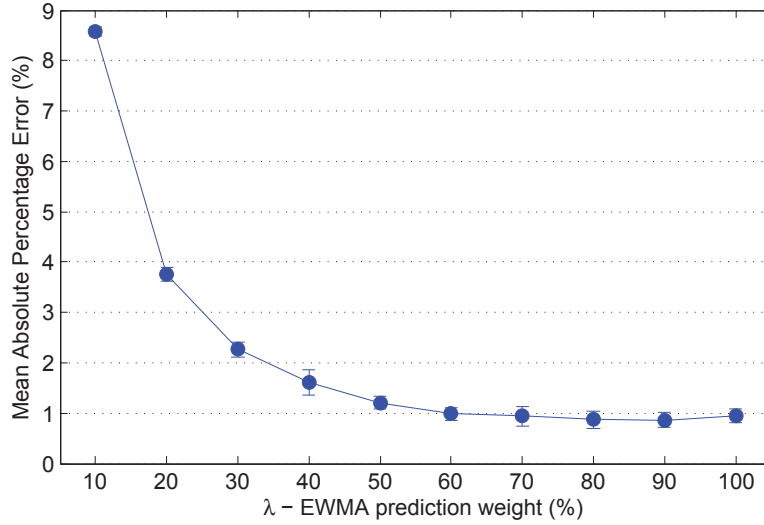


FIGURE 3.5: Effect of weight for EWMA on prediction error using static workloads

3.2.3 AEWMA

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 [67].

In multimedia and other dynamic applications a substantial transition can be observed [68]. The prediction error increases at the beginning of these transitions. The traditional EWMA algorithm has been modified to take these transitions into account. This new prediction approach is called the Adaptive Exponential Weighted Moving Average (AEWMA). Based on the work by Nemhhard [68], 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.

In order to determine where these transitions happen, the analysis from Table 3.1 was repeated. It was found that specifically for video applications, these happened on the I frames ⁶. The analysis was run on the same videos from Section 3.2.2.1 i.e. 3 videos of h.264 VGA resolution (640x480) run at 24 fps, for 30 seconds, run 100 times each. The process for doing the analysis of variation between grouped frames is shown in Figure 3.6, where a section of a sample video was taken to illustrate this. The steps are:

⁶For sake of uniformity, frame type names are mapped from video, i.e. I, P, B, to 1, 2 and 3 respectively

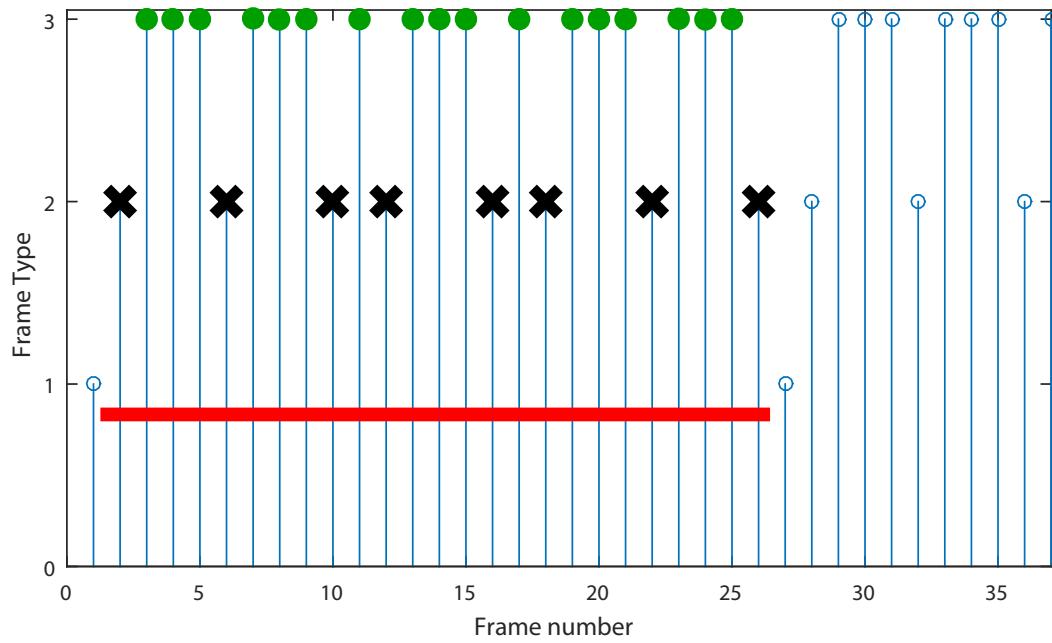


FIGURE 3.6: Sample of video with frame types 1, 2 and 3. Red line represents the group of frames of type 2 (cross) and 3 (green filled circles) between frames type 1 (blue hollow circles) used to calculate local standard deviation.

1. Identify frames type 1. In Figure 3.6, type 1 are the white circles.
2. Group frames around two type 1. They are represented by the red line.
3. Group frames inside the type 1 by frame type. Type 2 frames are black crosses. Type 3 frames are green circles.
4. Calculate standard deviation of the grouped type 2 and type 3.
5. Repeat until the video has been completed. Calculate the average of the standard deviations for type 2 and 3. Calculate standard deviation of type 1.

	Workload standard deviation (CPU Cycles)	Workload standard deviation grouped between frame type 1 (CPU Cycles)
No grouping	2.23×10^7	—
Frame types		
Type 1	1.68×10^6	1.68×10^6
Type 2	1.47×10^7	3.58×10^6
Type 3	1.93×10^7	4.25×10^6

TABLE 3.2: Comparison of variation of the workload with and without grouping into frame types

The comparison between variations is shown on Table 3.2. On one side, the frames are grouped by frame type, then its standard deviation calculated (as done in Section 3.2.2.1), and the procedure explained for Figure 3.6. This table shows the advantage of grouping frames between transition frames (frame type 1). With lower variations, predictions become more accurate. The transitions were found to be more convenient every frame type 1 given the lower variations (based on standard deviation). Therefore, it was determined the need to use AEWMA based on the Table 3.2, which shows a clear reduction in the prediction variation, which in turn accounts for more accurate predictions. Figure 3.7 shows the modification of λ on an application with 4 transitions.

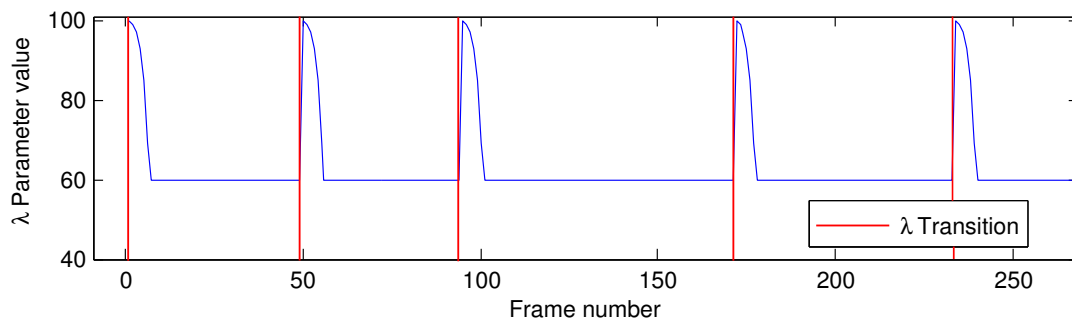


FIGURE 3.7: AEWMA λ parameter change at transitions

3.3 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*. The Algorithm 2 in Section 2.4.4 was modified to get integrated with Shepherd. The modified algorithm is shown in Algorithm 5. The modifications of the algorithm are:

- Determine the State based on the predicted workload coming from the Prediction Unit
- Given that the new State does not depend on the previous State, in Algorithm 2 line 7, the importance of future rewards is 0, so $\gamma = 0$, therefore the term $\gamma \max_{a'} Q(s', a')$ is removed.

Algorithm 5 Q-Learning algorithm, modified for Shepherd. Based also on Algorithm 4. Taken from [63]

```

1: initialise Q-Table
2: loop(for each epoch):
3:   obtain predicted_workload
4:   map predicted_workload to  $s$ 
5:   choose  $a$  (V-F) from  $s$  using  $\epsilon$ -greedy policy (Algorithm 6)
6:   take action  $a$ 
7:   ...wait for frame to finish...
8:   obtain  $r$  based on cost function (Figure 3.8)
9:    $Q(s, a) \leftarrow Q(s, a) + \alpha[r - Q(s, a)]$  (Figure 3.9)
10: end loop
```

As explained before (Section 2.4.4), Q-Learning creates a lookup table where knowledge is stored. The main objective of Q-Learning is to learn what to do (action, a) when a situation arises (state s). The lookup table is then a collection of the possible states with the possible actions to take for each state, and the accumulated reward for each state-action pair ($Q(s, a)$). These accumulated rewards represent the knowledge of the system. In this Q-Learning implementation, both the state and action spaces are discrete and finite. Given that the Q-Table is stored in memory, the resources are limited, therefore having a large state space is impractical. As the predicted workload

arriving is a 32-bit number (4,294,967,296 possible numbers), it has to be mapped to a smaller state-space (16 states were proposed). Therefore the first step after receiving the predicted workload, is to map it to a discrete state, workload-to-state mapping. After the state has been obtained, the Q-Learning starts to be executed.

Originally the Decision Unit has no knowledge of the system, therefore it requires to build the system model at run-time. The ϵ -greedy algorithm (Algorithm 6) provides a suitable form to build this model. The decisions are gradually taken, from exploring the possible choices, to exploiting the optimal choice for a particular State. The Decision Unit must start exploring decisions in different States to find the optimal (or most suitable) Action for a particular given 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 work is defined as the lowest V-F pair that fulfills the performance requirement.

Algorithm 6 ϵ -greedy strategy. Based on [8]

```

1: function CHOOSE_ $a(\epsilon, s)$ 
2:   generate random  $x$ 
3:   if  $x < \epsilon$  then                                     ▷ Exploitation
4:      $a \leftarrow \arg \max_a Q(s, a)$ 
5:   else                                                 ▷ Exploration
6:      $a \leftarrow$  random action
7:   end if
8:   if  $\epsilon < 1$  then
9:     increment  $\epsilon$ 
10:  end if
11:  return  $a, \epsilon$ 
12: end function

```

3.3.1 Cost Function

After the decision of the V-F setting has been taken, Shepherd sends the signal to the CPU frequency driver to change the Voltage and Frequency accordingly. The workload is then executed. After the workload has finished execution, it is necessary to review the V-F choice to learn how favorable was it for the application. This is done by gathering feedback of the system, and then it is evaluated in a *cost function*. In Shepherd, the feedback comes from the hardware in the form of execution workload, which is translated into the time it took to finish execution. This is then compared to the deadline to see how

energy efficient was the decision and how it affected performance. The main constraints and considerations for the design of the cost function of Shepherd are:

- Meet the deadline. As Shepherd intends to treat the environment as a Soft Real-Time system, this is the first priority. Given that it is impossible to ensure the deadline will be always met, it is tolerable to miss the deadline by a small margin.
- Save as much energy as possible, whilst meeting the deadline.
- As described in the environment where Shepherd runs (Section 3.1.1), the idle time cannot be accumulated, so it cannot be utilised to process the next frame.
- There is no Power Gating in the system, so as mentioned in Section 3.1.1 as well, the run-to-idle strategy cannot be applied.

Based on these considerations, the ideal situation is for the workload finished time $t_{finished}$ to be equal to the deadline $t_{deadline}$, so $t_{finished} = t_{deadline}$. Therefore, the cost function designed for Shepherd is:

$$r = \begin{cases} \frac{t_{finished} - t_{deadline}}{t_{deadline}} + 1 & \text{if } t_{finished} \leq t_{deadline} \\ \frac{t_{deadline} - t_{finished}}{t_{deadline}} & \text{if } t_{finished} > t_{deadline} \end{cases} \quad (3.3)$$

The cost function is represented in Figure 3.8. As an example, maximum reward is capped at 1. Four scenarios arise in this cost function (shown in Figure 3.8):

- (A) $t_{finished} \ll t_{deadline}$. The deadline was met but the system wasted energy being idle (there is no Power Gating). Therefore the reward is positive but relatively low.
- (B) $t_{finished} \leq t_{deadline}$. The workload was finished close to the deadline, with low idle time. The reward is positive and high. The highest reward is when $t_{finished} = t_{deadline}$
- (C) $t_{finished} > t_{deadline}$. The workload was finished after the deadline, but just missed. The reward is negative but small, becoming more negative depending on the margin of failure to achieve the deadline.
- (D) $t_{finished} \gg t_{deadline}$. The deadline was missed by a large margin. The reward is considerably negative.

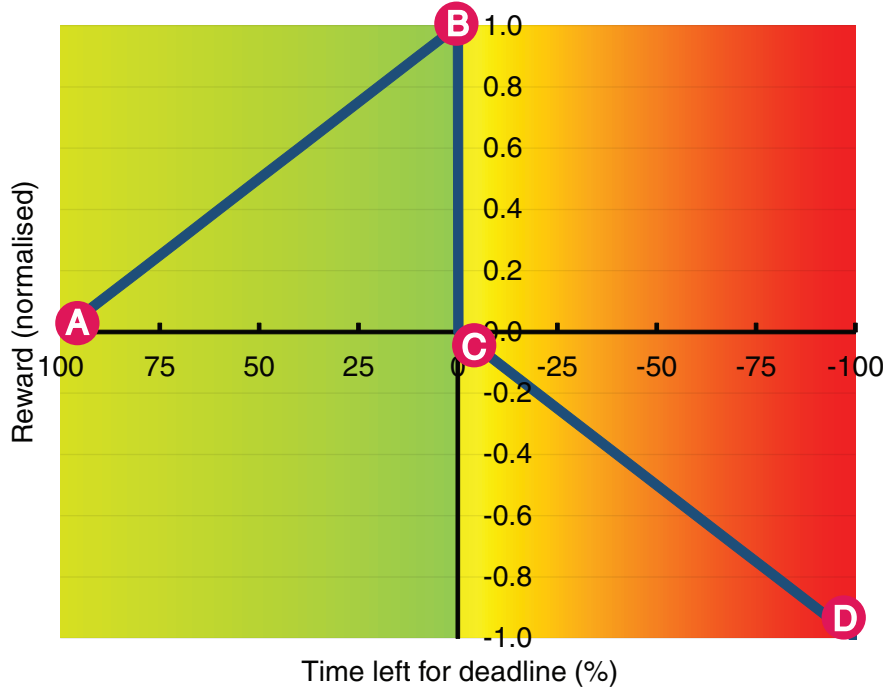


FIGURE 3.8: Cost function of Q-Learning algorithm for Shepherd. This graph shows the level of reward/punishment obtained for finishing the workload too early (A), very close to the deadline without surpassing it (B), finishing the workload shortly after the deadline (C) and finishing very late (D). Time left for deadline is calculated as $100 \frac{t_{deadline} - t_{finished}}{t_{deadline}}$

3.3.2 Learning Phase in Shepherd

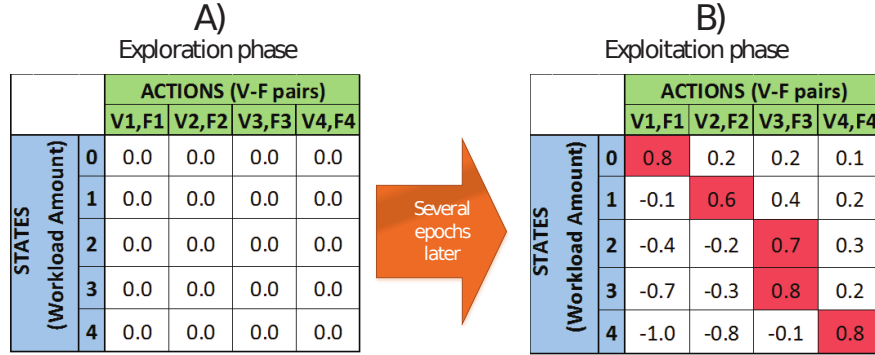
Learning is stored as values in a Q-Table, which in practise it is a lookup table with values corresponding to all State-Action pairs. At each decision epoch⁷, the decision taken for the last frame is evaluated; the reward or penalty calculated from the cost function is added to the corresponding Q-Table entry, thereby gaining experience on the decision. This is shown in Algorithm 5 line 9. The Q-table is updated then at the entry $Q(s, a)$, with the previous value plus an amount of the reward:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r - Q(s, a)] \quad (3.4)$$

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. In this case, the best action to take at a given epoch is the action with the highest accumulated reward. This phase of the algorithm is called the *Exploitation phase*. Figure 3.9 shows the evolution of the Q-Table.

⁷In reinforcement learning terminology, the interval at which the algorithm is triggered is known as decision epoch.

Initially, the values in the Q-Table are all zeros (Figure 3.9(A)); subsequently, in the exploitation phase, the best actions are determined (highlighted in red in Figure 3.9(B)).



A)
Exploration phase

		ACTIONS (V-F pairs)			
		V1,F1	V2,F2	V3,F3	V4,F4
STATES (Workload Amount)	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

Several
epochs
later

B)
Exploitation phase

		ACTIONS (V-F pairs)			
		V1,F1	V2,F2	V3,F3	V4,F4
STATES (Workload Amount)	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 3.9: Q-Table during A) exploration and B) exploitation phases. The red boxes represent the best Action for each State.

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 decisions⁸. 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.

Figure 3.10 shows an example of the evolution of the suitabilities of 4 different decisions. In this case the 4 different decisions are 4 V-F settings. They are the suitabilities across a particular workload state (State 4) of the Q-Table. In this example, the Q-Learning finds that the V-F pair of 800MHz is more suitable for State 4, running a particular application.

3.4 Discussion and Summary

This chapter presented the design of Shepherd, the RTM to be run together with the OS, learning to make decisions on future workloads by predicting them. After explaining the need for a more optimised approach to Run-Time Management in Chapter 2, Shepherd has been presented. The RTM is a cross-layer system that interfaces applications with the Power Management hardware, thanks to a modular algorithm with two major components, the Prediction and the Decision Unit. The environment in which the RTM operates has been described. This included the description of the environment as a cross-layer, with Application layer, OS layer and Hardware layer. The environment targets frame-based applications that have an implicit performance requirement and takes advantage of it, transforming the environment into a Soft RT system.

⁸Appropriate decisions are those that reduce the energy consumption, while satisfying the performance.

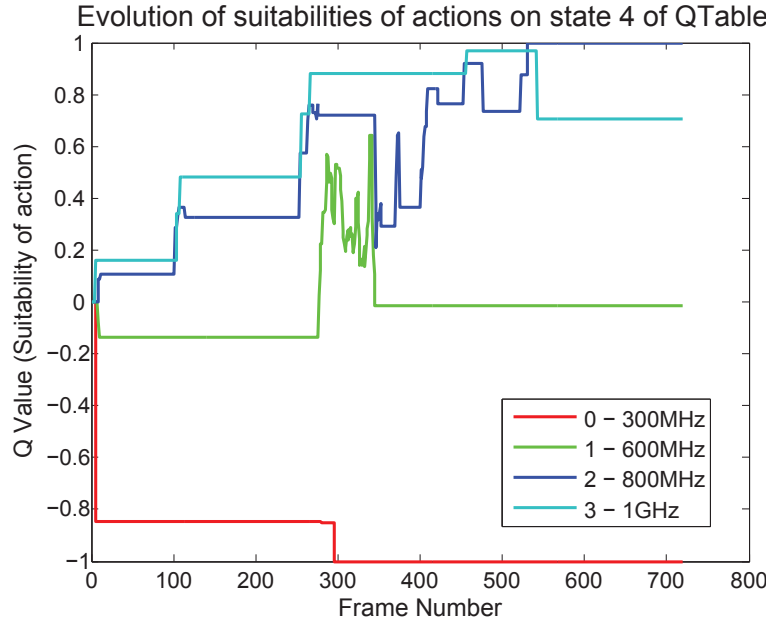


FIGURE 3.10: Suitability of actions for a particular state (state 4), defined by the Q Values of the different actions

Shepherd is presented as a system of two interconnected modules, the Prediction and the Decision Unit. It receives orders from the application, as well as annotations. The orders include the configuration instructions, the fps performance requirement and the number of workload types to be processed. The annotations are sent to trigger a frame end and a new frame start, as well as the frame/workload type. The Prediction and Decision Units then start working.

The purpose of the Prediction Unit is to estimate the workload ahead for the upcoming frames based on the workload history, using the EWMA prediction algorithm. The need for grouping the workload was explained as well, demonstrating less variation (standard deviation) within the groups as without them. Also, the optimisation of the parameters of EWMA was described. Having identified the shortcomings of having static parameters, AEWMA was introduced. As opposed to classic EWMA, the prediction parameters are adjusted at run-time, adapting to higher frequency variations in the workload.

After having predicted the workload, the Decision Unit started working. The main two components of the Decision Unit were the workload-to-state mapping, and the Q-Learning decision algorithm. The ϵ -greedy algorithm was used to allow for exploration and exploitation of the decisions to change the V-F setting. Exploration was used when the knowledge of the effectiveness of the decisions is not known, and exploitation when the Decision Unit has enough experience to know which decision to take at which situation (state). After the workload was finished, the algorithm proceeds to gather experience, by evaluating what the performance of the decision was.

Now that the RTM had been designed and optimised, it was necessary to implement it in a real device. Chapter 4 shows the implementation of the algorithm, as well as the presentation of the Shepherd API framework, which allows for applications to be transformed into soft real-time with power optimisation, becoming power-aware. Chapter 5 provides the results of the implementation, as well as the demonstration of the framework in some applications.

Chapter 4

Implementation of Run-time Manager

With Shepherd designed, it was decided to implement it for testing. This chapter explains the implementation decisions taken, justification on the implementation as a Linux Governor. Also the implementation challenges for the algorithm to perform at Operating System (OS) level. The use of an additional Linux kernel module is explained. Finally the design of a framework for integration in other applications is explored.

4.1 Implementation as Linux Governor

The placement in each layer has advantages and disadvantages:

Hardware The algorithm has no execution overhead for the system, so it can be expanded as needed. It also has less memory and execution limitations (e.g. Floating-Point Unit (FPU) operations), so the algorithm can be implemented as a hardware module, collecting data directly from the performance counters. This would also enable the Run-time Manager (RTM) to be only awake when needed (with timers), using Power Gating (PG) for the rest of the idle time. The main disadvantage is the implementation and testing time increase, including the complexity of making the algorithm able to be upgraded. The other obvious disadvantage is that legacy devices and System-on-Chips (SoCs) cannot benefit from the algorithm.

Application The algorithm is relatively easier and faster to implement, as there are no memory/execution limitations, and there are inter-process communication tools that allow the annotations to be passed to the RTM. One disadvantage is that the activation of the RTM depends on the task scheduler of the OS, therefore being

unable to ensure a Real-Time response. Moreover, the inter-process communication may not be able to give the timing guarantees needed for the algorithm to perform as expected.

- OS** The algorithm module provides better timing guarantees than at Application level, as the module is activated both when the application requires it, plus the use of other OS level modules (Dynamic Voltage and Frequency Scaling (DVFS) and performance counter modules) is executed faster because there is no change of context (between Application and OS).

It was decided for the RTM to be implemented at OS level. It was done in this fashion given that Linux already provides DVFS controllers at OS level called Linux Governors. The Governor is a Loadable Kernel Module[69], a piece of code that extends the functionality of the OS. It has access to hardware functions i.e. changing the power state, or Voltage and Frequency (V-F) pair. The Governor provides an interface for the user to manipulate the Governor's parameters.

Shepherd Governor was implemented in Linux Kernel 3.7.10, which also includes other Governors such as Ondemand, Conservative, and Performance. Shepherd provides a distinct difference compared to other Linux Governors apart from the intrinsic functionality, and that is the interface to connect applications at user level with the kernel module itself. The Governor was implemented in the BeagleBoardxM (BBxM) [9], which contains the DM3730 SoC made by Texas Instruments [11]. The SoC is a single core 32-bit ARM Cortex-A8 architecture.

Figure 4.1 shows the implementation of the Governor as a group of kernel modules, with the arrows pointing out the communication with User space and the hardware layer. It shows the most relevant functions implemented in the kernel module. It is important to mention that the decision to implement Shepherd as a Linux Governor provides an analogy of the cross-layer design of the algorithm of Figure 3.2. In this way, the algorithm maps directly to the RTM design.

The main difference with Figure 3.2 is the inclusion of the Performance Counters Kernel module, which translates the Performance Counts from the CPU Registers into the Real Workload required by Shepherd. The functions performed by the Shepherd Kernel module and the Performance Counters Kernel module are listed, as well as the Application Programming Interface (API) functions required to be included in the application. This Linux Governor is composed of two kernel modules, the Governor itself and the Performance Counters module (Section 4.4). The Governor module is composed of three activities: the algorithm that decides the V-F settings, the communication with the application in User space and the communication with the Performance Counters module.

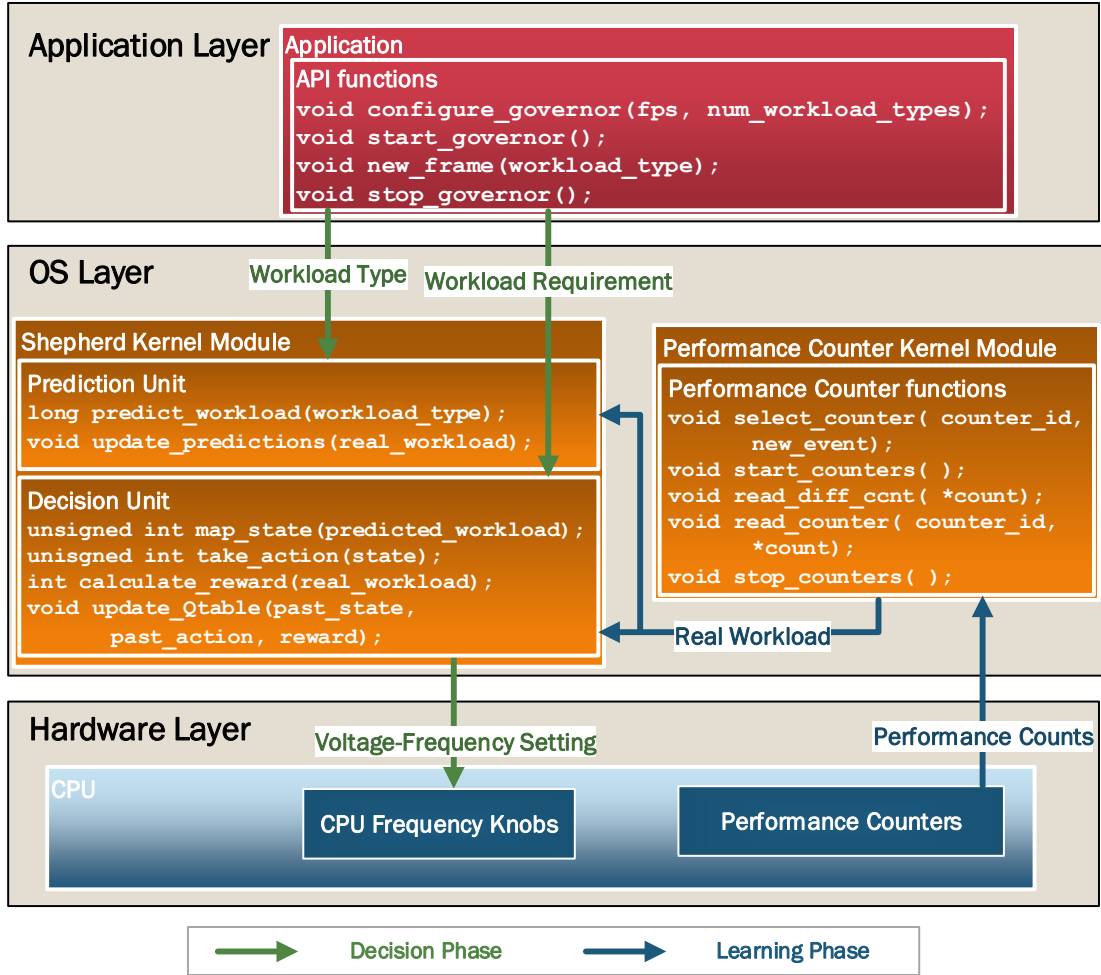


FIGURE 4.1: Shepherd governor implementation

4.1.1 Restrictions and optimisations for Linux Governor Implementation

As stated by Bovet and Cesati [70] and Maurer [59], in Linux kernel mode the use of the FPU, and other coprocessors e.g. Single-Instruction-Multiple-Data (SIMD), NEON, MMX, etc. is quite restricted and rarely done. The main reason is that the FPU (and others) may be in use by the applications (processes). In order to use the coprocessors these must store their user-mode contents in temporary registers, get the kernel processing data loaded into them, get cleared and get their user-mode contents restored. At kernel mode, this is a time demanding process that may supersede the benefits of using the coprocessors in the first place. Therefore it was necessary to optimise the mathematical operations used in the Shepherd algorithm.

The most significant optimisations done to the implementation as a kernel module are:

- No use of decimal point. The algorithm requires some integers to be multiplied with weights e.g. the prediction weight, which is $0 \leq \lambda \leq 1$. Instead the weights

are done as $0 \leq \lambda \leq 100$. To avoid overflow, the integers are shifted 4 bits before multiplication, then shifted back. The error produced by the right shifts is minimal compared to the values being handled (usually greater than 1 million). This is seen in Listing 4.1.

- Less use of division and multiplication. Right/left shifts were used where possible to avoid divisions and multiplications.
- Simplify use of random number generation. Random numbers are a key element for the decision unit. They are used in the exploration/exploitation decision and for exploration itself. Every run of Shepherd there is at least one random number required. Instead of using random engines, the random numbers are obtained each frame by using the 8 Least Significant Bits (LSBs) of the CPU cycles measured. Figure 4.2 presents the distribution of these random numbers with a mean of 127.4071 (minimum 0 and maximum 255) using a 4 minute video, of 5760 frames. This provides enough evidence to use them as the simple random engine.

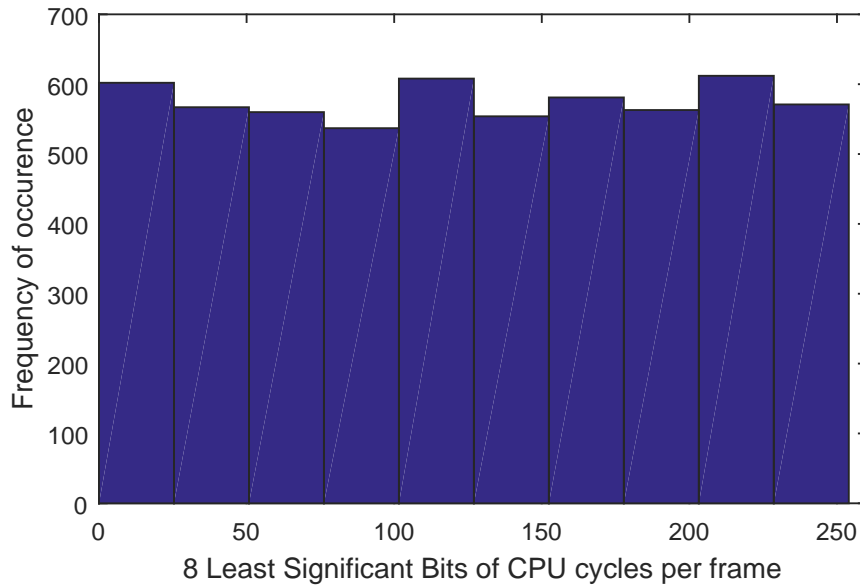


FIGURE 4.2: Random number distribution generated from taking the 8 LSBs from the CPU cycles measured per frame, over 5760 frames.

4.2 Implementation of Prediction Unit

The Prediction Unit was implemented in both the normal Exponential Weighted Moving Average (EWMA) and the upgraded Adaptive Exponential Weighted Moving Average (AEWMA). As demonstrated in Section 3.2, the EWMA produces more accurate predictions when grouping frame types together i.e. doing workload prediction of similar

frame types. Therefore, an array of weighted averages was allocated, with as many elements as frame types exist in the application. The frame types are also called workload types in this context. The EWMA algorithm has three main functions:

- `reset_predictions()` Cleans the weighted average array.
- `configure_prediction_unit(num_workload_types)`. The Prediction Unit allocates the necessary number of arrays based on the number of workload (frame) types for that particular application.
- `predict_next_workload(workload_type)` The algorithm executes Equation 3.1 for the given workload type, so the prediction for the next frame is the weighted average of the previous frames of that type.
- `update_prediction(new_workload)` The new weighted moving average is updated with the workload just gathered for that workload type, and the update is done based on Equation 3.2.

The main difference between EWMA and AEWMA is the prediction weight λ , which is kept constant in the former and used as a variable in the latter. Therefore, the AEWMA needs an extra function:

- `update_lambda_weight(workload_type)`

In this function the prediction weight λ is modified if certain conditions are met. The function models the behaviour explained in Section 3.2.3. When a transition occurs the variable λ is set to a top value λ_{TOP} , and decays to its original value λ_0 :

1. If the workload type is of the transition type, the prediction weight is ramped up i.e. $\lambda = \lambda_{TOP}$ to give more importance to recent workloads.
2. If the workload is not a transition and $\lambda > \lambda_0$, λ is decreased with an exponential behaviour. For this exponential decrease to be computationally efficient is done dividing by 2, simply shifting λ one bit. If $\lambda < \lambda_0$ then $\lambda = \lambda_0$.
3. If the workload is not a transition and $\lambda = \lambda_0$, do nothing.

The parameters of the Prediction Unit were selected from Section 3.2, with values of λ_{TOP} and λ_0 these being 100% and 60% respectively.

4.3 Implementation of Decision Unit

The Decision Unit was implemented using the modified Q-Learning algorithm discussed in Section 3.3.

As mentioned before, the Q-Learning's knowledge is stored in the Q-Table, made of States (rows) and Actions (columns). In this implementation, the States are defined as the amount of workload to be processed. The Actions are the possible choices of the algorithm, which in this context they are the V-F pairs. The number of Actions (V-F options) for this platform is 4. Given that the performance counters from the platform analysed have 32-bit registers, the maximum number of Central Processing Unit (CPU) cycles countable is $2^{32} - 1$, or 4,294,967,295. The number of States was decided to be 16, with each State representing a range of workloads e.g. State0 is from 0 to 1×10^8 cycles, State1 from 1×10^8 to 1.5×10^8 cycles, etc. The values for each limit were selected based on the occurrence following several runs of the Governor. The Q-Table is declared as a 2-D array of signed 16-bit integers of 16x4. The number of States and the workload ranges can be optimised for reducing memory overheads and improve power consumption. Listing 4.2 shows an auto-generated Q-Table after running the governor for 4 minutes doing video decoding. The optimisation of these parameters (number of states, ranges of each state) is left as future work.

The main functions executed by the Decision Unit are:

- `initialise_decision_unit(fps_requirement)` Clear the Q-Table. Add performance constraint to the cost function.
- `map_state(predicted_workload)` Based on the predicted workload identify the current Q-Table State. Returns the State number.
- `take_action(state)` Based on the Q-Table and the current State select the Action to be taken, which is V-F pair. The exploration/exploitation algorithm (Section 3.3) is used in this function.
- `calculate_reward(real_workload)` This function is called after the frame has been processed. It uses the cost function to calculate the reward or penalty for the decision taken.
- `update_qtable(reward)` The Q-Table is updated accumulating the reward for the State and Action taken.

When a new frame is about to start, the Decision Unit receives the predicted workload for that frame. The predicted workload is a 32 bit integer that represents the CPU cycles the next frame is going to consume. As mentioned before, the Q-Table of the Decision Unit consists of 16 States, each State representing a range of CPU cycles. The predicted workload is then mapped to one of the 16 States, defining which value will be used.

Given the restrictions for implementing algorithms in a Linux kernel module (Section 4.1.1), the exploration/exploitation algorithm was designed using the simple random number. The exploration/exploitation ratio ϵ is defined as an 8 bit integer. The higher the value of ϵ , the more probable exploitation is going to happen and vice versa. Therefore, at the start of the application, ϵ is set to 0. The maximum value of ϵ is 255. Every frame, ϵ is incremented, gradually reducing exploration in favour of exploitation. The determination to explore/exploit is done obtaining the simple random number $r(n)$ and comparing it to ϵ , if $\epsilon \leq \text{rand}_{8bit}$ then explore, else exploit.

When exploring, another simple random number rand_{2bit} is generated to take the Action. Given that only 4 Actions (4 V-F pairs) are possible in this particular platform, only 2 bits are used from $s(n)$, from a range of 0 to 3. The 2 bit random number is then used as the exploring decision.

When exploiting, the knowledge of the Q-Table is used to decide the next Action. Given the identified State based on the predicted workload, the Action is selected on that particular State. This Action is the most suitable one, based on the experience of the Decision Unit itself. The implementation of the selection of the most suitable Action is explained in Section 4.3.2.

After the decision has been taken for either exploring/exploiting, the Governor sends a `cpufreq_change()` request to the CPU. After the V-F has been changed, the Governor becomes dormant, and control is returned to the application, where the workload will be processed. After the frame has been executed and a new frame is ready to be executed, the Shepherd Governor is called again to learn from the previous decision. The implementation of the learning is explained in the next section (Section 4.3.1).

4.3.1 Implementation of Learning Phase in Shepherd

The new State has been identified and the Action has been taken. After the frame is executed, the Decision Unit must learn about its latest choice. The Governor emits a request to the Performance Counters Module (Section 4.4) to obtain the CPU cycles of that particular frame. The real deadline time $t_{deadline}$ is calculated based on the performance requirement pr and the time overhead it takes for the Shepherd algorithm

to process $t_{overhead}$ (Equation 4.1). The time it took for the workload to process $t_{workload}$ is calculated with the processor cycles $CPUCYCLES$ used for the workload, retrieved from the Performance Counter Module and the frequency f at which the workload was executed (Equation 4.2). The reward r is calculated as the difference of $t_{deadline}$ and $t_{workload}$ (Equation 4.3). A $t_{workload}$ greater than $t_{deadline}$ yields a negative reward. In order to improve execution time of the Governor and reduce overhead, the two major differences with the originally designed algorithm (shown in Figure 3.8 and Equation 3.3) are:

- In the design the reward is normalised to 1, with rewards being a fraction, and highest reward being 1. This was done differently for two reasons, fractions imply the use of floating point numbers, which are not allowed to be used naturally in a kernel module; and given that division is a CPU cycle costly function[71, 72], it was avoided. Therefore, unsigned integers were used for calculating the reward.
- Given that the reward is not normalised to 1, and to reduce the use of arithmetic operations, the larger the yielded r i.e. $t_{deadline} \gg t_{workload}$, the lower the reward was in reality. Therefore, the maximum reward was implemented to be 0 i.e. $t_{deadline} = t_{workload}$. When exploiting, for the current state $s_{current}$ the `take_action(state)` function takes the current action $a_{current}$ to be the lowest positive accumulated reward $Q(s_{current}, a)$, so $a_{current} \leftarrow \arg \min_a Q(s_{current}, a)_{\geq 0}$.

$$t_{deadline} = \frac{1}{pr} - t_{overhead} \quad (4.1)$$

$$t_{workload} = \frac{CPUCYCLES}{f} \quad (4.2)$$

$$r = t_{deadline} - t_{workload} \quad (4.3)$$

As an example, the performance requirement pr is set to $23.976fps$, equivalent to $41708\mu s$. Giving worst case estimations for Shepherd Governor execution time, the overhead $t_{overhead}$ is $0.5ms$, larger than the overhead recorded (presented in Section 5.4). The real deadline is then $41208\mu s$ (Equations 4.4 and 4.5). Receiving $CPUCYCLES = 25659200$ from the Performance Counter Module, the time $t_{workload}$ is calculated in microseconds using frequency f in MHz like the example in Equation 4.6. The reward r is then computed as the difference $t_{deadline} - t_{workload}$, yielding a reward of $r = 9134$, the time units are removed.

$$t_{deadline} = \frac{1}{pr} - t_{overhead} = \frac{1}{23.976fps} - t_{overhead} \quad (4.4)$$

$$t_{deadline} = 41708\mu s - 500\mu s = 41208\mu s \quad (4.5)$$

$$t_{workload} = \frac{CPUCYCLES}{f} = \frac{25659200cycles}{800MHz} = 32074\mu s \quad (4.6)$$

$$r = t_{deadline} - t_{workload} = 41208\mu s - 32074\mu s \quad (4.7)$$

$$r = 9134 \quad (4.8)$$

The calculated reward is then sent to update the Q-Table. The Q-Table is updated as in Section 3.3. The implementation is shown in Listing 4.1. The Learning Rate α (ALPHA) is set to 40%. The shifts are done to avoid overflow in the variables.

```
temp1 = ALPHA * (reward >> 4); //ALPHA = 40
temp2 = (100 - ALPHA) * (QTable[previous_state][previous_action] >> 4);
temp3 = temp1 + temp2;
QTable[previous_state][previous_action] = (temp3 / 100) << 4;
```

LISTING 4.1: Q-Table learning code

4.3.2 Action Suitability for Exploiting Learning

As explained before, the highest accumulated reward is the lowest positive value on that particular State, which is selected as the ‘best’ action. As an example, the Listing 4.2 is a generated Q-Table from a run of Shepherd on the BBxM. As a reference, let the new State be 3. Table 4.1 shows the Q-Values of State 3. The most suitable Q-Value is $Q(3, 1)$, for State 3 use Action 1 (marked in blue in Table 4.1).

QTABLE				
ACTIONS				
	0	1	2	3
STATES				
0	15872	15856	0	0
1	0	0	0	0
2	0	-2528	6272	6688
3	-15520	3712	11856	11216
4	-27728	-3584	8480	13104
5	-19808	-7440	8560	12832
6	-16304	-7040	4944	9792
7	-15232	-3024	4736	13312
8	0	0	0	0
9	-8400	0	0	0
10	0	0	0	0
11	0	0	0	0
12	-16640	0	0	0
13	-352	0	0	0
14	0	0	0	0
15	-8240	0	0	0

LISTING 4.2: Auto-generated Q-Table for a video decoding application.

State	Actions			
	0	1	2	3
3	-15520	3712	11856	11216

TABLE 4.1: Q-Values of State 3 of Q-Table from Listing 4.2

4.4 Performance Counters Module

Some ARM Cortex processors provide Performance Counters as coprocessors for monitoring the cores inside the SoC. In order to be able to get information from the Performance Counters for calculating the workload, the ARM cores provide registers to control and access the data[73]. The ARM Cortex-A8 CPU (used in this research) contains a performance counter coprocessor called Performance Monitoring Unit (PMU). This Unit contains memory mapped registers accessed by writing directly to them in assembly language. The BBxM used in this research provides direct access to the performance counters, whereas other platforms require the use of Debug tools such as JTAG to enable them.

The Linux kernel provides an interface to access these performance counters at user level called *perf*[74], but to access the counters at kernel level it presented a more challenging task. The interface was not trivial to be controlled and it was never found the proper way to run the interface without errors. Approaches exist that provide APIs with functions at user level, one called *PAPI*[75, 76] and the other *perfmon2*[77]. The drawback is that these functions are user level only, defeating the purpose of having a kernel level functional module.

For control and use of the performance counters at kernel level, the two options available were to use assembly code inline with the governor code, allowing for direct access whenever it needed to request data from the registers; or to implement a custom Performance Counters Module to take care of the monitoring tasks leaving the governor exclusively to do the run-time management. It was decided to implement the latter, appealing to a more modular design approach. This enables scalability and portability. For example, the complexity of porting the governor to another platform with different hardware specifications and counters having a separate kernel module for performance monitoring. The Cortex-A8 uses the coprocessor C9 and can simultaneously count 4 events from a list of 50 events, plus the CPU cycle counter. The Cortex-A9 [78] provides 6 counters for any of the 58 events available.

The functions implemented by the performance counter module are shown in Figure 4.1. They were implemented in assembly code required for writing directly to the performance counter registers. The functions are:

- `select_counter(counter_id, new_event)` Assign a new event (from the 50) to one of the 4 available counters.
- `start_counters()` The counters are reset. Start the counters.
- `stop_counters()` Stop the counters.
- `read_counter(counter_id, *count_variable)`. Read one of the 4 available counters and store the value in `count_variable`.
- `read_diff_ccnt(*count_variable)`. Read the CPU cycle counter, calculate the difference with its value from the last time it was read and store the value in `count_variable`.

Given that the task done most in the governor regarding performance counters is to read the CPU cycles (done every frame), the computation of the difference between the last frame until this one was calculated inside the module, and then accessed with `read_diff_ccnt()`.

In the Linux kernel, functions and variables can be exported as symbols[59, 61, 69]. This is done to enable kernel modules to access functions and variables from another module. To create the symbol the macro `EXPORT_SYMBOL` is used on the function/variable to be exported. When the kernel module is loaded, its address is sent to a table that holds the addresses of all the global kernel variables [59]. In this case, the governor code had to import the header file of the counter module to access these functions. This creates kernel module dependency. To debug the designed module it was exported using `sysfs`[61]. This enables its use from user level, exporting some parameters as a filesystem.

4.5 An API for Shepherd

As mentioned in the Introduction, Pang et al. [37] present a survey of software developers stating the lack of knowledge of best practices and tools to produce energy efficient software. Moreover, only around 10% of the respondents measure the energy consumption of their project. There is an opportunity to allow the other 90% to provide energy efficiency to their code without the need to analyse/measure consumption.

Given the availability of a power efficient governor that takes the application into account, it was decided to enhance the governor to allow developers to produce energy efficient applications. As explained in Section 3.1, Shepherd takes its decisions based on the workload and the performance constraints and annotations coming from the application. Therefore, it is necessary for the application to pass these constraints and annotations via an interface. An API was designed together with the Shepherd governor to facilitate this communication.

As explained before, the three signals that Shepherd 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 The objective of Shepherd is to get the application to behave as Real-Time (RT), so its performance requirement is the deadline, in terms of a particular deadline per application 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 Signals that alert the RTM when the application has started its main loop, and when finishes it.

These three pieces of information need to be sent to the RTM to work efficiently. These signals are implemented in the framework. The API therefore provides the following functions:

- `configure_governor(fps,num_workload.types)` This provides the performance requirement (`fps`) and some information for the governor (`num_workload.types`) to annotate the workload to be processed and improve the governor accuracy. This is the governor setup.
- `new_frame(workload_type)` Every new frame tells the governor which type of workload is going to be processed.
- `start_governor()` This starts the governor.
- `stop_governor()` This stop the governor.

These functions are shown in Figure 4.1. In the figure it can be seen the green arrow from API functions connects to the Governor using *ioctl*. The *ioctl*[59] is a system call used in Device Drivers to access hardware devices from User-level. This is done by opening a virtual I/O file that opens the interface. The procedure is then to write to this virtual file the command to be executed in the kernel, with the possibility of sending data, in this case the parameters of the API function i.e. frames-per-second (`fps`), etc. The Governor is ready to accept these system calls whenever they happen.

The next subsection shows the general design flow for changing an application to use the RTM.

4.5.1 Design flow for application adapted to use Shepherd Governor

The general approach to integrate an application with the Shepherd governor is presented here:

1. **Identify application suitable for Shepherd** The main objective of Shepherd is to provide Real-Time performance on an application. This means that the application needs to have a *timing requirement* for it to be optimised. Examples of this are multimedia, including video decoding/encoding, audio decoding/encoding, image processing.
2. **Identify segments of the application** The Governor works best when the application is divided in segments. These segments include the beginning and end of the application. As the Real-Time is given by processing a segment as a frame, ideally the main processing task should run in a loop where each iteration represents a frame. Video decoding and image/audio encoding present this behaviour.
3. **Treat segments as frames and annotate them** Frames are divided by workload types. If the variation in workload amount among the frames is high, it is suggested to split them in groups of similar workload amounts. These constitute workload types.
4. **Set performance constraints and annotations** Decide on the target fps for the application, this will be the performance constraint for the Governor. Also, count the different workload types of the application.
5. **Annotate application** Add the API functions to the application. The functions `configure_governor()` and `start_governor()` should be set at the beginning of the application (before the main loop). `new_frame()` should be added at the beginning of each iteration of the main loop. `stop_governor()` should be added at the end of the application.

Let us have a generic application e.g. FFT benchmark[12]. This application is to be modified to use Shepherd. In order to be used, the application will run for 100 times in a loop, changing to three different window sizes (therefore three workload types). The application contains a Program Header which is represented by the preamble before running the main demanding task. This includes flag parsing, parameter definitions, memory allocations, etc. The FFT itself is the Program Main Loop, and at the end of the program, the Program Footer represents memory freeing, file saving, etc. This is represented in Figure 4.3 A).

In order to use Shepherd, the code using the Shepherd framework must be added, which enables communication to it. Figure 4.3 B) shows the sections of the application that need the addition of the interfacing code to talk to the RTM. In the Program Header the performance constraints are sent together with the Start signal. In the Program

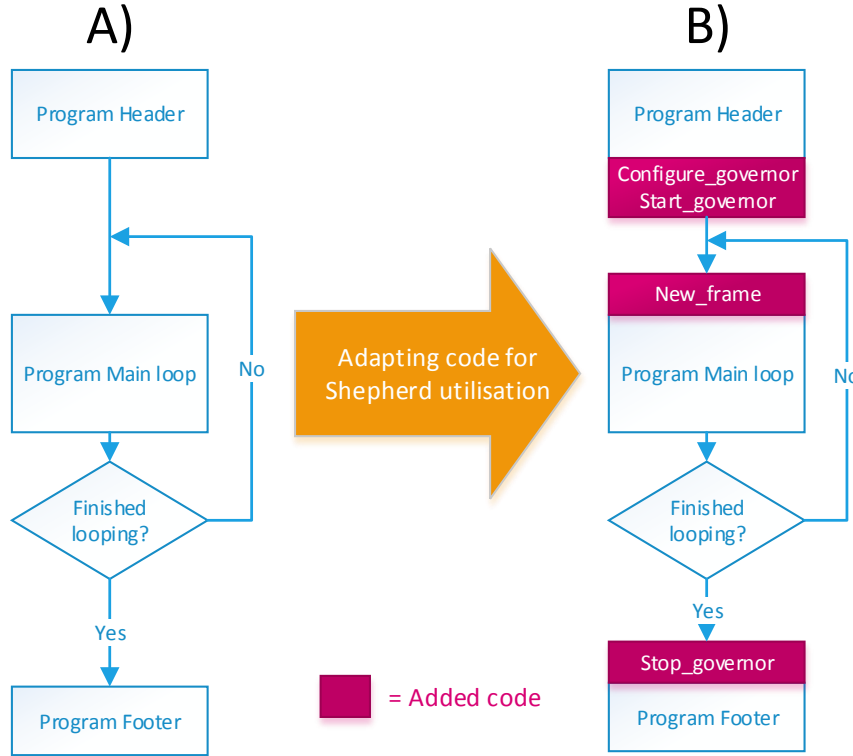


FIGURE 4.3: Adaptation of program code for utilisation of the Shepherd Run-Time Manager

Main Loop, the ‘New Frame’ signal is sent to Shepherd, which includes the annotation of which ‘workload type’ is to be processed. After the Main Loop has finished, the Stop signal is sent to alert Shepherd to end listening for new frames. After the additions of code, Shepherd starts to learn the different behaviours of the workload and adjusts the V-F pairs accordingly. The framework code to be added is designed to be minimal and completely unobtrusive to the rest of the application code.

4.6 Discussion

This chapter provided a thorough description of the challenges faced when implementing the RTM Shepherd, and the solutions to achieve it. The overview of the RTM was explored, together with the implementation of the algorithm and the components needed for it to function. It was designed as a Linux Governor, which runs at Kernel level, providing a fast response with the added complexity of programming it. The Prediction and Decision Units were implemented, providing the techniques to have efficient code resulting in smaller processing overheads. The performance counters module was explained, which was required to be a separate module to allow for portability of the Governor. The Shepherd API framework expands the possibilities of software developers to improve the energy efficiency of their applications, moreover it provides a black box for the programmer not to deal with the complexity of the power management itself. Next chapter explores the validation of the Shepherd Governor and the API.

Chapter 5

Results

This chapter presents the results generated by running the Shepherd Governor in the BeagleBoardxM (BBxM) platform.

5.1 Experimental Setup

The experiments of the Run-time Manager (RTM) were realised using the BBxM [79], which has the DM3730 [11] System-on-Chip (SoC) from Texas Instruments (TI). The results show that the proposed governor adapts to varying workloads and achieves significant energy savings when compared to the other available governors, whilst providing a desired performance. This chip contains a single-core ARM Cortex-A8 processor (together with a DSP and a GPU). The specifications of the CPU are listed on Table 5.1. The CPU supports 4 Frequencies.

Figure 5.1 shows the BeagleBoardxM platform with the peripherals connected: HDMI to DVI cable connected to a monitor for video decoding, power connector, serial and ethernet connection for remote control using Secure Shell (SSH). The BBxM runs Linux 3.7.10+x10 kernel, with the Ubuntu 12.04 distribution on top. No GUI (like GNOME or GTK+) was running with Ubuntu, to reduce the memory consumption and isolate the experiments as much as possible. The procedure for every experiment was as follows:

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 5.1: DM3730 Specifications (ARM Cortex-A8 processor) [11]

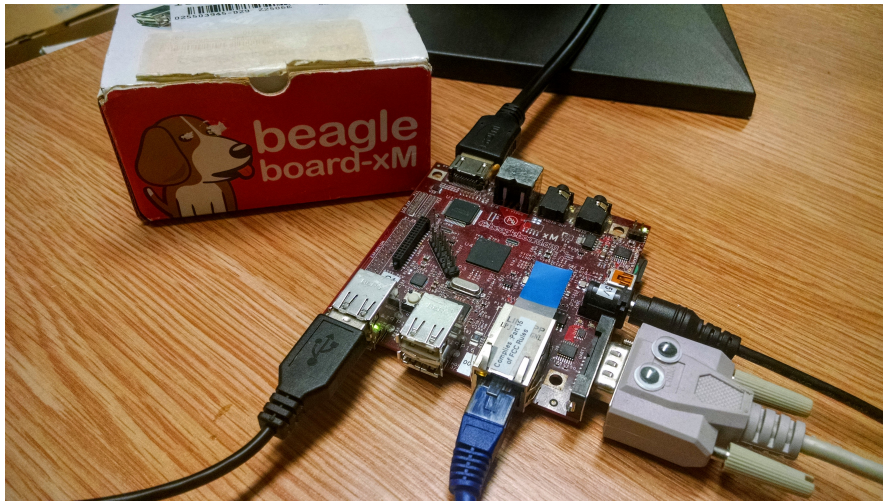


FIGURE 5.1: BeagleBoard-xM[9] development board used for running the experiments of Shepherd

1. Verify that the Power connector is providing power to the board.
2. Press the Reset button on the board. Wait for the board to boot. It finishes booting up when the login screen appears.
3. Enter login and password and wait for the cursor to appear.
4. Run the governor setup script. This was necessary to configure the governor to use for a particular experiment. Verify that it finishes setting up the desired governor
5. Run the experiment script. This script would setup the experiment, and the setup includes setting up the file store for the output files (with performance and energy measurements). It finalises starting the experimental run. This starts the application on subject. Specifically for Shepherd, the application has been modified to configure and trigger the start of governor processing.
6. Wait for the application to finish its run. After it finishes running, the data generated should be at the file store.

Figure 5.2 shows an example of the BBxM running an experiment using video decoding. In order to compare the efficiency and performance of Shepherd, it was tested against other Linux Governors. The other 5 available governors are: *performance*, *powersave*, *ondemand*, *conservative* and *userspace* (600 and 800). The comparison is done with the different Linux Governors:

powersave This is a static governor (does not change frequency over time) that is set at lowest possible frequency. For the BBxM, this frequency is $300MHz$.

userspace600 This static governor runs at $600MHz$.

userspace800 This static governor runs at $800MHz$.

performance This static governor runs at maximum frequency ($1GHz$). It is the most power consuming governor.

ondemand This is a dynamic governor that varies its frequency over time, by looking at the processor demand. The governor looks on a time window what the idle time was, and if it goes below a threshold the frequency is ramped up to the highest setting, gradually going down after every sample time to get between a low and high threshold.

conservative This is a dynamic governor similar to *ondemand*, with the difference that instead of ramping to the highest frequency, the governor gradually increases the frequency setting until it reaches the desired idleness per sample time.



FIGURE 5.2: Video Decoding experiment using Shepherd Governor running on BeagleBoard-xM

5.2 Case Study: Run-Time Manager for Video Decoding

In order to test Shepherd, *Mplayer* was selected, which uses the *FFmpeg*[80] library of video codecs. The application was slightly modified in order to send its performance constraint as well as the task annotations to the governor. As mentioned before, these

performance constraints are sent using *ioctl*. It is important to mention that the governor is not exclusively designed for video applications, but as video applications present an inherent performance constraint (frames-per-second), they are suitable for these tests.

Different video codecs and video sizes have been tested. The video codecs are:

- h264 vga resolution (640x480)
- h264 qvga resolution (320x240)
- h263
- MPEG-4
- MPEG-2

In total 5 videos per codec have been run. Each video runs for 720 frames, which is equivalent to 30 seconds running at 24 frames-per-second (fps). The videos vary in their workloads. In the 5 different video types there exist 3 different frame types, I,P and B, which represent different *workload types* in the Governor. This approach is similar to Choi et al. [34], in the sense that they also distinguish the different frame types. Energy and power consumptions were calculated using the values from Table 5.1, which show the processor at normal runtime. For example, for frame x if processor was running at 1GHz then its power consumption for that frame would be 877.01mW.

The data of the videos run was stored in Comma-Separated Values (CSV) files at runtime, and then imported to MATLAB for analysis for obtaining the results. Appendix A shows an auto-generated data file from a video run. The data fields included are: Frame Number, Frame type (and its numeric value), frame time in microseconds, deadline pass/fail, predicted and real workload, and the frequency at which the frame was run.

5.2.1 Real time behaviour of Shepherd

Figure 5.3 shows the comparison in time of both Shepherd and the ondemand Governor, displaying the performance and the power consumption of both governors. Performance in this case is known as the closeness to the target fps stated by the video, which in this particular case are 23.98 fps. The nature of the video decoder (for memory purposes) is to decode frames in real time with a one frame buffer, therefore only one frame is decoded and displayed in an epoch. This means that the slack is not accumulative, and there may be only slack from the decoded frame. So, the maximum achievable frame rate for this application is the target frames-per-second i.e. 23.98 fps. If a frame is decoded in more time than aimed for, there is a performance loss. Power consumption is measured in Weight (W), which represent the amount of instantaneous power used during a particular

frame decoding. Hence, the aim of Shepherd is to reduce the area below the green curve (of power consumption) with minimal reduction of the performance target (23.98 fps).

It can be seen from Figure 5.3 that overall in terms of power consumption Shepherd always stays below ondemand (except for periods between 13-22 and 118-124), meaning a consistent save in energy. It can be seen also that these energy savings produce some performance losses, one clearly stated as the *learning phase* and the other ones being some exploration decisions. The learning phase (in orange) shows the period where Shepherd is learning the optimal power states for these workloads, this by taking random decisions. After 115 frames the Run-time Manager has found the optimal states and therefore starts taking better decisions. After this learning phase is finished, the system goes into exploitation mode, “exploiting” the most adequate decision for every situation. Because the workloads are dynamic, Shepherd still explores during the exploitation phase, with a more sporadic occurrence, costing small performance losses.

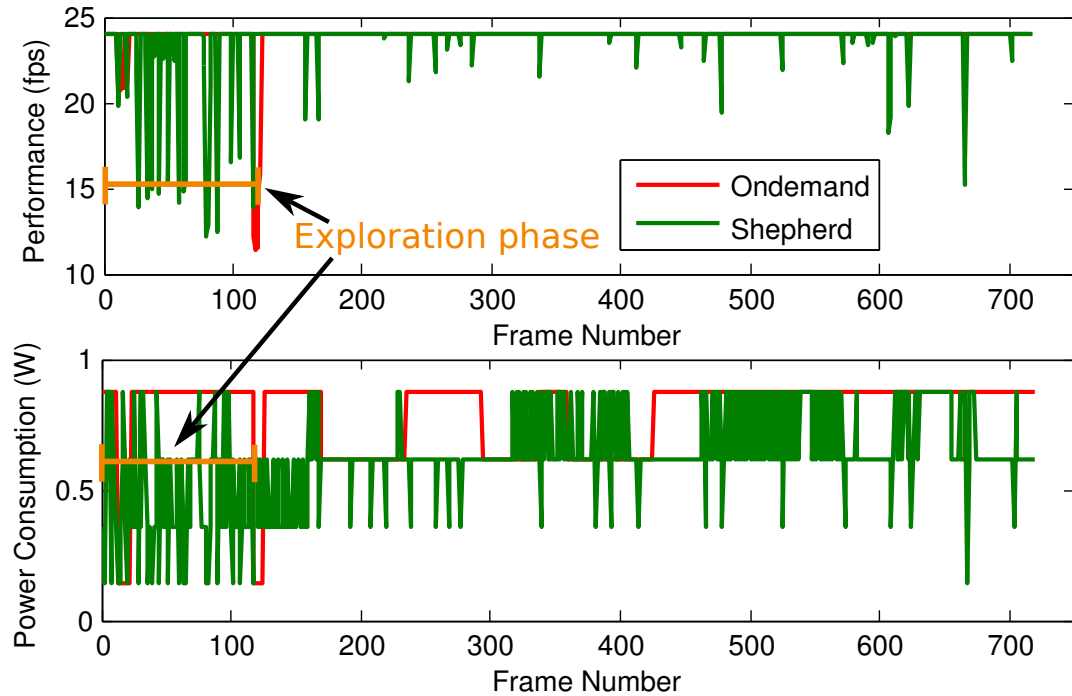


FIGURE 5.3: Performance and Power Consumption of the governors Shepherd and Ondemand for an H.264 Video

5.2.1.1 Shepherd on H.264 VGA

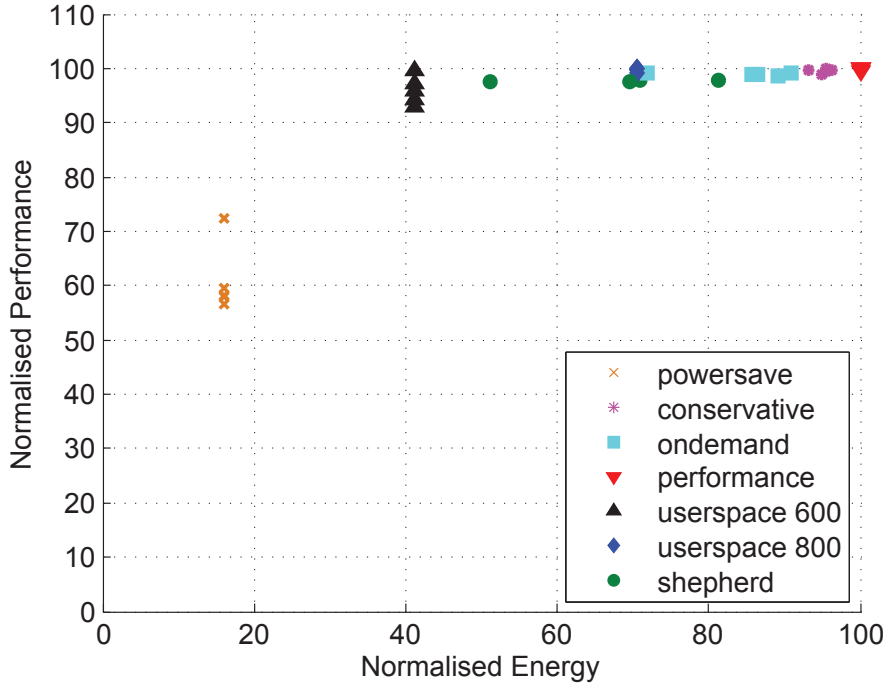


FIGURE 5.4: Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H264 VGA Video

Figure 5.4 shows a pareto graph that compares the energy consumption versus the performance. To facilitate the reading of the figure, both axis were normalised from 0 to 100. The 100% for energy represents the maximum energy consumed by the Performance governor. It is clear that the Performance governor should have the maximum consumption, given that it is always running at $1GHz$. The 100% in normalised performance represents the maximum performance in terms of the number of frames in which the deadline was achieved. Having a point in 100% normalised performance means that the governor achieve 100% of the deadlines. Each point represents the average frame rate achieved per video. There are 5 points per governor, representing the 5 videos run in simulation. It is deduced that the ideal point is the top left corner of the figure, with 0% energy consumption and 100% performance, and therefore is the target of Shepherd.

It can be seen clearly that the Powersave governor provides the least power consumption at the expense of around 60% performance. On the opposite side the Performance governor achieves 100% of the deadlines, but consumes the most amount of energy. Comparing the static governors, the Userspace 600 provides the best compromise of energy and performance, with more than 90% performance and significant power savings. It could be argued that Userspace 600 is the optimal governor to use, but that decision was not known a priori. From the dynamic governors, Shepherd clearly provides better power savings at a small penalty on performance when compared to Conservative and

Ondemand. The power savings obtained by Shepherd surpass 20% compared to the other dynamic governors.

5.2.1.2 Shepherd on H.264 QVGA

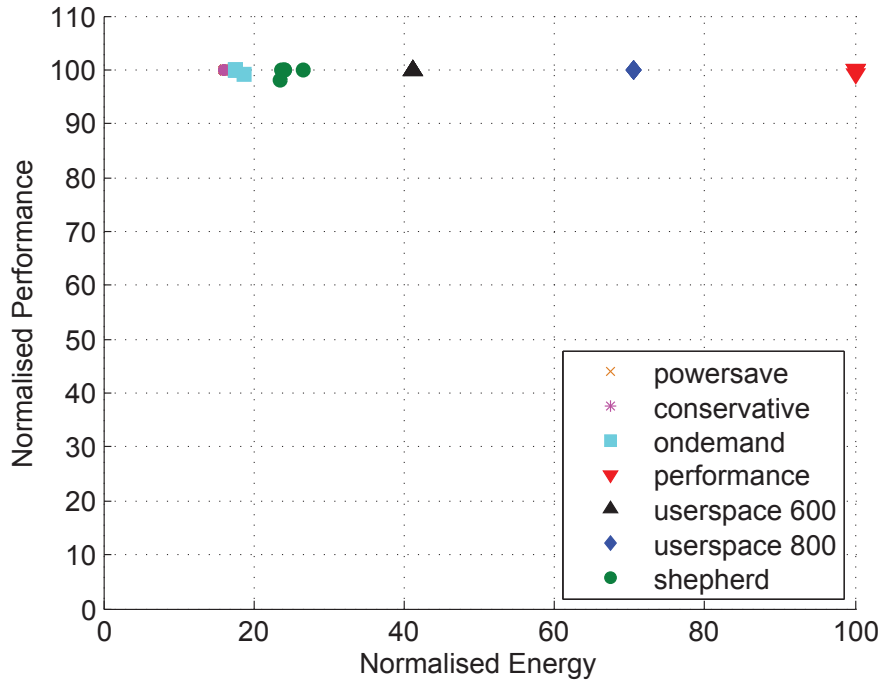


FIGURE 5.5: Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H264 QVGA Video

From the H.264 QVGA experiment shown in Figure 5.5 it is clear that the processing demand is reduced compared to VGA. It is logical as QVGA resolution (320x240) is lower than VGA (640x480), so the effort is reduced. This shows then that conservative and ondemand provide more power savings than Shepherd, for the reason that their algorithm reacts when the processing demand is very high (less than 20% idle time), and given the reduced workload, this never happens. There are few Voltage and Frequency (V-F) changes in Conservative and Ondemand, so these governors are in the 300MHz setting for most of the time.

Instead, Shepherd consumes more energy because it has an exploration phase, so the high frequencies are also visited during this period.

5.2.1.3 RTM on H.263

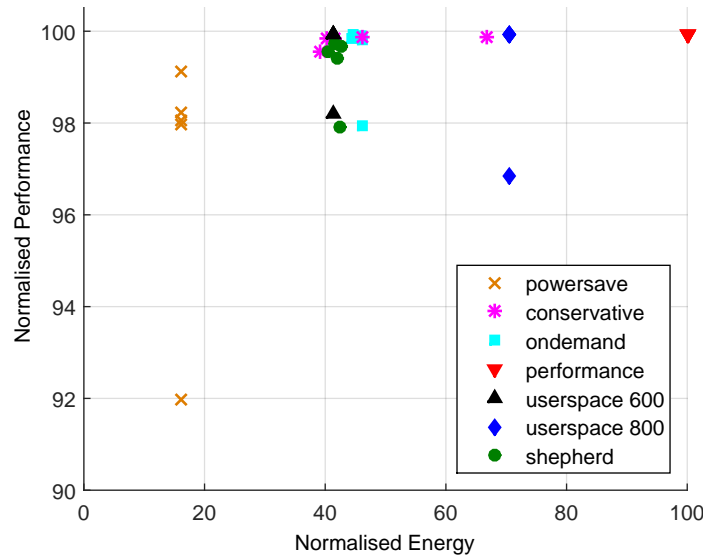


FIGURE 5.6: Pareto graph comparing different governors vs. Shepherd by means of energy and performance running H263 Video

Figure 5.6 provides a close up compared to the two previous pareto figures, where it is observable that every dynamic governor performs in a similar manner. Overall the average energy consumption can be seen slightly higher for Shepherd.

5.2.1.4 Shepherd on MPEG2

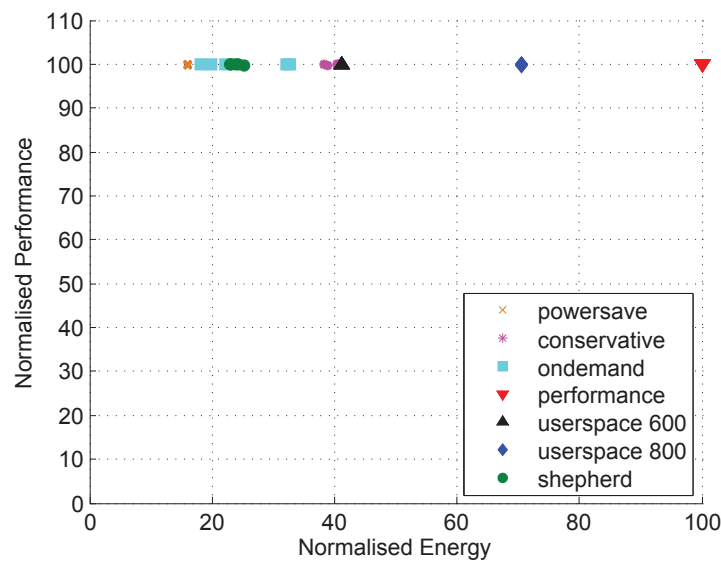


FIGURE 5.7: Pareto graph comparing different governors vs. Shepherd by means of energy and performance running MPEG2 Video

Figure 5.7 provides a similar performance like Figure 5.5, given that the workload demand is reduced.

5.2.1.5 Shepherd on MPEG4

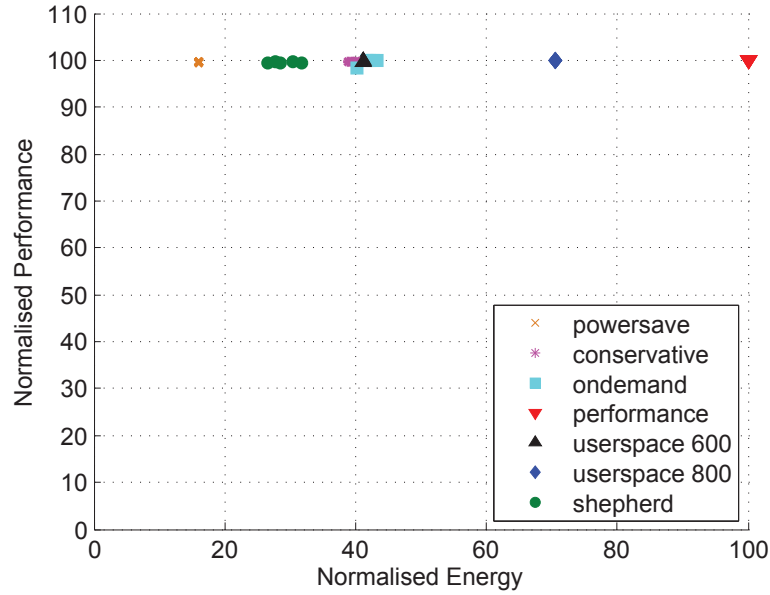


FIGURE 5.8: Pareto graph comparing different governors vs. Shepherd by means of energy and performance running MPEG4 Video

In Figure 5.8 it can be seen that Shepherd found more energy savings, where ondemand and conservative spent most of the time in the 600MHz level.

5.2.2 Governor comparison

A comparison of overall power efficiency has been done with Shepherd and the other Linux governors. Table 5.2 provides the comparison, in which the numbers are the averages of the normalised performance and energy consumption of every video, using the different decoders. An extra comparison is done with a longer video, run 4 minutes. VGA and QVGA refer to H.264 decoder with VGA (640x480) and QVGA (320x240) respectively, and VGA_4mins refers to an H.264 VGA video run 4 minutes, as opposed to 30 seconds (for the rest of the videos). Every video codec was run in VGA resolution (except for H.264 QVGA as previously stated).

5.2.2.1 Comparison versus Dynamic Governors

Overall it is shown that Shepherd provides a similar performance against dynamic governors (1.9% lower performance when lowest in vga), but provides marginal energy savings in H.264 VGA and H.264 VGA_4mins, with 29.5% and 11.7% less energy consumption than Conservative and Ondemand respectively for VGA_4mins.

In MPEG4, Shepherd provides 11% and 13% less energy consumption than Conservative and Ondemand respectively. In H.263 energy consumption is 5% and 3.5% lower,

		Static Governors (Frequency in MHz)				Dynamic Governors		Shepherd
Video Decoder		300	600	800	1000	Ondemand	Conservative	
VGA_4mins	Energy	16.1	41.2	70.5	100.0	81.5	99.3	69.8
	Performance	60.1	93.3	98.5	99.7	99.6	99.6	98.4
VGA	Energy	16.1	41.2	70.5	100.0	84.8	95.0	68.7
	Performance	60.9	96.1	99.7	99.8	98.9	99.6	97.7
MPEG4	Energy	16.1	41.2	70.5	100.0	42.0	40.0	29.0
	Performance	99.7	99.9	99.9	99.9	99.6	99.8	99.5
H.263	Energy	16.1	41.2	70.5	100.0	45.5	46.8	41.8
	Performance	97.1	99.6	99.3	99.9	99.5	99.8	99.3
MPEG2	Energy	16.1	41.2	70.5	100.0	25.0	39.8	23.9
	Performance	99.9	99.9	99.9	99.9	99.9	99.9	99.8
QVGA	Energy	16.1	41.2	70.5	100.0	17.8	16.1	24.4
	Performance	99.9	99.9	99.9	99.8	99.8	99.9	99.5

TABLE 5.2: Comparative table showing the paretos of energy and performance averages of every Linux governor compared to Shepherd. Numbers are normalised to 100. Ideal performance is 100, and ideal energy is 0.

respectively. In MPEG2 energy consumption of Shepherd is 15.9% and 1.1% lower respectively.

In H.264 QVGA however, Shepherd consumes more energy than Conservative and Ondemand, 8.3% and 6.6% more. This happens because of the nature of Conservative and Ondemand algorithms. These governors start scaling the V-F setting when the minimum idle time threshold is reached. The workload of H.264 QVGA decoding is significantly lower compared to higher resolutions, where on average there are 3.2956×10^7 cycles per frame in H.264 VGA versus 1.5943×10^7 in H.264 QVGA. Therefore Ondemand and Conservative cross the minimum idle time threshold few times per video in Ondemand and 0 times in Conservative, so the increment in the V-F setting is done on average 1.2 times in Ondemand and 0 times in Conservative. As mentioned in Section 5.2.1.2, the increase in energy consumption of Shepherd happens due to the exploration phase of the Decision Unit, where Shepherd selects the V-F settings at random to find the optimal.

5.2.2.2 Comparison versus Static Governors

Static Governors provide a fixed V-F setting for the whole workload. For this reason, energy consumption of each Static Governor remains constant (100% for Performance Governor, 70.5% for Userspace800, 41.2% for Userspace600, 16.1% for Powersave)¹. This makes them impractical when the workload is unknown. If the workload could be profiled offline, then the V-F setting could be selected, either per frame or one for the whole application. For this reason, in some applications a Static Governor may perform better than Shepherd i.e. in MPEG2 Shepherd consumes 7.8% more energy than Powersave (300 MHz), but this does not apply to every application i.e. in H.264 VGA the

¹The Static Governors and their frequencies are Performance= 1GHz, Userspace800= 800MHz, Userspace600= 600MHz, Powersave= 300MHz

Powersave Governor (300 MHz) gives a performance of 60.9% whereas Shepherd gives 97.7%. Therefore, it is unrealistic to consider Static Governors as the ideal choice given their lack of flexibility and offline profiling requirements to work. It is then shown that Shepherd gives an overall better Energy-Performance trade-off than both the Dynamic and Static Governors available.

5.3 Shepherd Application Programming Interface (API) results

In order to demonstrate its versatility, the Shepherd API was used on different benchmarks, setting arbitrary workloads for two different MiBench benchmarks[12]. These are “FFT” and “inverse FFT” (or iFFT). The applications themselves were modified using the API for notifying the governor. In order to provide a frame-oriented approach, the applications are run in a loop, with each time the application is run constituting a cycle. These experiments were run for 700 frames (cycles). In order to test the adaptivity of Shepherd to changes in the performance constraints, the workload was kept constant and the performance target (objective) was changed for every experiment. Both “FFT” and “iFFT” were run with four different constraints, 8, 10, 12, and 16 fps. Tables 5.3 and 5.4 show the results of Shepherd and the other governors performance running the FFT benchmark, with the different performance constraints. Tables 5.5 and 5.6 show the results of the performance *versus* energy for each performance constraint, for the iFFT benchmark.

It is clear that the larger the performance constraint, the faster the workloads need to be finished, so for a higher performance constraint, higher frequencies are expected to be required. This can be seen with the static governors, and particularly userspace 800, which reaches the first three targets but fails to reach 16 fps in Table 5.4. It can be seen upon further inspection that for static workloads a static frequency proves to be optimal for each target. The problem would be finding which is the optimal frequency. Figure 5.9 shows the real time behaviour of Shepherd, which also has the learning phase (shortened to 36 frames), then starting to do exploitation, plus little exploration. These exploration decisions plus the learning phase contribute to an expected loss in performance and slightly more power consumption than ideal, penalties paid for being online learning.

Going back to Tables 5.3, 5.4, 5.5 and 5.6, the performance loss discussed before can be seen, as Shepherd does not average the target but slightly less, also reaching a high improvement in energy consumption. It is fair to say that as dynamic governors do not have intrinsic performance constraints, and that FFT and iFFT are computation demanding processes, the governors run normally at highest frequencies, consuming large amounts of energy. By looking at both tables it is clear to see that iFFT has a lower

workload than FFT, and Shepherd takes advantage of this, reducing the energy consumption further down e.g. Shepherd at 12 fps. It is clear then that upon knowing the optimal frequency the adjustments the governor should stay on that particular frequency. Shepherd is always assuming that workloads are dynamic, therefore exploration is necessary even after enough confidence has been reached on the decisions.

Governor	Objective: 8fps		Objective: 10fps	
	Performance (fps)	Normalised Energy	Performance (fps)	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
shepherd	7.5	40	9.7	67

TABLE 5.3: FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective

Governor	Objective: 12fps		Objective: 16fps	
	Performance (fps)	Normalised Energy	Performance (fps)	Normalised Energy
powersave	4.5	16	4.5	16
userspace 600	9.1	41	9.1	41
userspace 800	12.0	70	12.1	70
performance	12.0	100	15.1	100
conservative	11.9	98	14.8	98
ondemand	12.0	100	15.1	100
shepherd	11.9	98	14.5	97

TABLE 5.4: FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective

Governor	Objective: 8fps		Objective: 10fps	
	Performance (fps)	Normalised Energy	Performance (fps)	Normalised Energy
powersave	6.2	16	6.2	16
userspace 600	8.0	41	10.0	41
userspace 800	8.0	70	10.0	70
performance	8.0	100	10.0	100
conservative	8.0	97	10.0	97
ondemand	8.0	100	10.0	100
shepherd	7.9	40	9.5	41

TABLE 5.5: Inverse FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective

Governor	Objective: 12fps		Objective: 16fps	
	Performance (fps)	Normalised Energy	Performance (fps)	Normalised Energy
powersave	6.2	16	6.2	16
userspace 600	11.9	41	12.4	41
userspace 800	12.0	70	16.0	70
performance	12.0	100	16.0	100
conservative	11.9	97	15.8	97
ondemand	12.0	100	16.0	100
shepherd	11.4	68	14.3	96

TABLE 5.6: Inverse FFT benchmark [12] performance vs. energy results. The governors aim to fulfil the objective

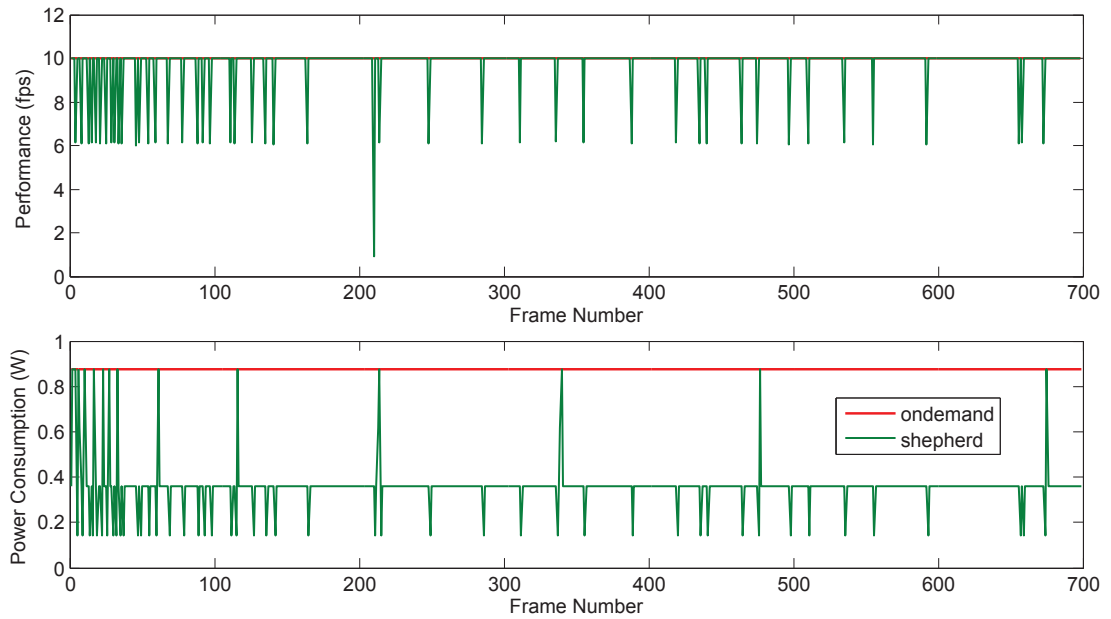


FIGURE 5.9: Performance and Power Consumption of the governors Shepherd and Ondemand for iFFT benchmark at 10 fps constraint

5.4 Overheads

The implementation of the algorithm as a Linux governor has negligible overhead. The BBxM's lowest frequency is 300MHz, and the algorithm takes $11.5\mu s$ to compute (from the time of the system call to the end of the decision epoch). The overhead including the frequency change 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 Shepherd governor uses $14kB$ of memory and does not use dynamic memory allocation for the Q-Table.

5.5 Discussion and Summary

This chapter has shown how Shepherd was implemented as a Linux governor, and also the implementation of the framework as an API was explained, based on the design presented in Chapter 3. The implementation included the validation of the prediction and adaptation algorithms, furthermore analysing the optimal values of the weight factor for the predictor. The Run-time Manager was first validated to work with dynamic workloads such as videos, and then compared with the other Linux Governors available. Results show that Shepherd is able to adapt and with slight performance losses, it can save large amounts of energy. It was also shown that for a longer video, energy savings increase, reaching around 15% savings *versus* the ondemand governor and around 30% savings against maximum frequency, with only 1.5% performance loss.

After comparing with different video codecs, the Run-time Manager was shown to work for newly created applications, presenting the case study of modifying an application to use the governors API. Added to the benefits mentioned, the performance overhead of running the governor is negligible. Next chapter presents the final conclusions, adding a detailed future work to be carried for the final thesis.

Chapter 6

Conclusions

Given the current need for more power efficient devices in the presence of highly demanding workloads, research has been driven towards minimising the energy costs of these workloads. The effort was taken in this work to design a cross-layer Runtime Power Manager, in order to address these required minimisations. Shepherd was proposed as a Runtime Power Manager that takes advantage of the opportunities presented by the applications to save energy. The energy savings are achieved by reducing the system Frequency (and Voltage) when possible, working on the analysis of the application at run-time to detect periods of lower workloads.

Shepherd shows the need for integrating predictive and adaptive Machine Learning techniques to optimise power consumption. The technique was done targeting single core Dynamic Voltage and Frequency Scaling (DVFS), managing an acceptable impact in performance. The predictive techniques aim to foresee the workload arriving at the next frame, and the Exponential Weighted Moving Average (EWMA) filter was implemented for its simplicity in terms of operation and memory usage, and high precision. Experimental results show a prediction error of around 3% for dynamic workloads such as video decoding.

The adaptive Machine Learning technique implemented is the Q-Learning algorithm, which starts learning the model of the system by exploring different decisions, rewarding good decisions and penalising bad ones; after that starting to exploit the most convenient decisions. The Q-Learning algorithm was used because it provides an online approach to making optimal decisions, added to this is its simplicity in terms of operation and memory usage. This means that no previous offline analysis is needed in order to determine the optimal decisions.

The cross-layer approach was presented as a standard framework to achieve better savings in power, emphasising the needed feedback from hardware. The advantages of using this Runtime Power Manager are that it is always trying to optimise for the best power

consumption, and the penalty paid is reduced. It is also important to note that the overheads in memory and time by the normal operation of the Governor are negligible, therefore ensuring that the Operating System (OS) is oblivious to the running of the Governor. Experimental results show that Shepherd takes around $12\mu s$ in the lowest frequency setting to execute, whilst the time normally taken for a video application per frame is around $40ms$.

The framework was designed to provide a platform for creating/modifying applications that may require performance constraints, in order to achieve power savings by means of the usage of the OS. The framework is designed so the application developer can control the Run-time Manager, and advice of further changes in the system. The design flow was presented, where it showed how an application can be adjusted to provide the annotations needed to select optimal values of the system.

Based on Section 5.2, the implementation of the Run-time Manager shows that although sometimes selecting a particular Voltage and Frequency (V-F) pair may seem as a more convenient choice, the user (and the other Governors) is unable to know beforehand which setting is the more appropriate to take. It is important to note that most of the performance losses presented by Shepherd are due to the exploration phase the Run-time Manager undertakes. For video applications Shepherd was shown to be a more effective choice than the dynamic governors *ondemand* and *conservative* in most cases: H.264 VGA, H.263, MPEG2 (better than conservative) and MPEG4. Energy savings achieved reach around 30% with a 1.5% performance loss for H.264 video of 4 minutes. For applications modified to use the framework (FFT and inverse FFT) being a static workload, Shepherd presents savings around 60% of energy for low demand workloads. For high demand workloads Shepherd aims to reach the objective, missing some deadlines due to the learning phase.

Overall, the Run-time Manager provides a more robust technique for embedded systems powered by an Operating System that demand high efficiency with high performance requirements.

6.1 Future Work

The work presented in this document is further driven into several directions:

Optimisation of predictions As seen on Figure 3.4, depending on the video and the experiment, the λ parameter may be more convenient at certain settings. This presents an opportunity to adjust this parameter online to generate more accurate predictions independent on the application. Artificial Neural Networks (ANNs) present a robust and efficient alternative for regression and time series predictions, and given the reduced complexity they may be suitable to be used as the predictor.

A review on more optimisation algorithms is needed to determine which alternative provides a better online parameter adjusting.

DPM - DVFS Interplay Shepherd currently does DVFS on the main CPU core, assuming only high demand workloads arrive. When the device is not in use, Dynamic Power Management (DPM) opportunities arise that enable the CPU to go to deep sleep modes. Also, as shown by Bhatti et al. [62], DPM and DVFS may be used together on a particular workload when knowing the Worst - Case Execution Time (WCET) and the deadline. This presents an opportunity to enhance the Run-time Manager to use DPM to further reduce power consumption when using general purpose applications.

Heterogeneous Power Management The Run-time Manager designed at the moment controls the frequency power knobs of a single-core CPU. Nowadays System-on-Chips (SoCs) are equipped with several cores including both CPU cores and HW accelerators, such as DSPs, GPUs, media codecs and transceivers. This provides an opportunity to enhance the Run-time Manager to take full control of the whole SoC by controlling the power states of all the internal modules. Normally power manageable accelerators are equipped with Power Gating features[47], so the Run-time Manager may be able to use DPM techniques on these accelerators.

Given that Shepherd works in a single core environment with a single application, three scenarios arise as the expansion of the idea: multiple applications running on single core, one application running on multiple cores, and multiple applications running on multiple cores. The scenarios and their needs for new research to be conducted are described as follows:

Multiple applications running on single core Currently Shepherd works on a single application running, using a Real-time constraint needed from the application. The Frequency Mapper (FM) implemented takes into account the uncertainty of the environment, such as OS overheads, other applications running in the background, and interprets them as noise. The Run-time Manager may be modified to provide with a more complex power management alternative that sees these background processes, or even other applications running simultaneously, in order to take power management decisions in a more realistic environment. The main component required for this task is an adaptive scheduler with a similar algorithm to Shepherd, that obtains the deadlines of every application running on it. The scheduler should be Soft Real-Time (RT) oriented to accommodate every deadline. Also, knowing the uncertainty of the workload to be processed, the scheduler must

have a predictive algorithm i.e. Adaptive Exponential Weighted Moving Average (AEWMA). The resulting design is a Soft RT scheduler with the Shepherd algorithm that controls DVFS.

Single application running on a multicore environment Nowadays SoCs consist of several CPU cores to run more efficient tasks. This creates several power management opportunities, where the Run-time Manager may decide whether to run the different cores at a particular speed, or to shut down unused ones, also trying to balance the load. The Run-time Manager may be enhanced to cope with these opportunities. The use of multicore devices is tightly coupled with integration to the task scheduler, as the Run-time Manager must have control of where the tasks are allocated to do efficient power management on the cores. The approach for Shepherd on multicore would have a new key element: the prediction/decision arrays are defined as the application energy saving profile, saved in RAM. This means that Shepherd creates a profile for the application to be run, which essentially contains the prediction and decision models of the application. Normally applications running in multicore are moved around the different cores for execution being allocated by the scheduler. Therefore, once the application model has been created, it can be used in each processor where the application is running, given that it is a similar core type. In case the new core running the application is different (in heterogeneous computing), the application profile should be rebuilt. Furthermore, the application profile may be saved in non-volatile memory for use next time the application is run.

Multiple applications running on multicore Similar to the previous approach, an application energy saving profile should be created per application, and loaded together whenever each application starts/continues running. In this approach, the scheduler must be Soft RT as the one proposed when running multiple applications, and where each of them has its own deadline.

Overall, the future work of the three scenarios described can be summarised in two needs: a Soft RT scheduler coupled with the Shepherd algorithm, and an application energy saving profile per application running, saved in RAM or in non-volatile memory.

Bibliography

- [1] Jprice. Low Power Design Guide, 2009. URL <http://www.powerforward.org/media/g/completeguide/default.aspx>.
- [2] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low power methodology manual: For system-on-chip design*. Springer Verlag, 2007. ISBN 9780387718187. doi: 10.1007/978-0-387-71819-4. URL http://books.google.com/books?hl=en&lr=&id=C4yy1NF0QwUC&oi=fnd&pg=PR14&dq=Low+Power+Methodology+Manual+For+System-on-Chip+Design&ots=H3k-GURCo6&sig=BH22eS1sSunDdHXw5gQYkN_x9gY.
- [3] N H E Weste and D M Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson Education India. ISBN 9788131762653. URL <https://books.google.com/books?id=XumCXbbaUJ0C>.
- [4] Primetime px user guide, June 2009.
- [5] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37:195–237, September 2005. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/1108956.1108957>. URL <http://doi.acm.org/10.1145/1108956.1108957>.
- [6] Luca Benini, Alessandro Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000. ISSN 1063-8210. doi: 10.1109/92.845896. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=845896>.
- [7] Michael Moeng and Rami Melhem. Applying statistical machine learning to multicore voltage & frequency scaling. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 277–286, New York, New York, USA, 2010. ACM. ISBN 9781450300445. doi: 10.1145/1787275.1787336. URL <http://portal.acm.org/citation.cfm?doid=1787275.1787336><http://dl.acm.org/citation.cfm?id=1787336>.
- [8] Lucian Busoniu, Jelmer van Ast, and Robert Babuska. Reinforcement Learning, 2010.

- [9] BeagleBoard. BeagleBoard-xM Rev C System Reference Manual, 2010. URL http://beagleboard.org/static/BBxMSRM_latest.pdf.
- [10] Gene Sally. *Pro Linux Embedded Systems*. Apress, 2010. ISBN 9781430272274. URL <http://ebooks.cambridge.org/ref/id/CB09781107415324A009>.
- [11] Texas Instruments. DM3730, DM3725 Digital Media Processors Datasheet, 2011. URL <http://www.ti.com/lit/ds/sprs685d/sprs685d.pdf>.
- [12] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench : A free , commercially representative embedded benchmark suite The University of Michigan Electrical Engineering and Computer Science. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001. doi: 10.1109/WWC.2001.15.
- [13] A Carroll and G Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on ...*, 2010.
- [14] Aaron Carroll and Gernot Heiser. The Systems Hacker’s Guide to the Galaxy Energy Usage in a Modern Smartphone. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, volume 9300, pages 5:1—5:7, 2013. ISBN 978-1-4503-2316-1. doi: 10.1145/2500727.2500734.
- [15] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3471 LNCS, pages 165–180, 2005. ISBN 3540297901. doi: 10.1007/11574859\12.
- [16] Michelle Bailey. Infrastructure Convergence: The Integration of Technology to Lower Cost and Improve Business Response IDC OPINION, November 2009. URL http://ftp.hp.com/pub/c-products/solutions/IDC_Converged_Infrastructure_White_Paper.pdf.
- [17] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, 2006. URL http://scourge.fr/mathdesc/documents/kernel/linuxsymposium_procv2.pdf#page=223.
- [18] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do Nothing, Efficiently. *Proceedings of the Linux Symposium*, pages 119–125, 2007. URL <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:cpuidle+-+Do+nothing,+efficiently...#0>.
- [19] Tajana Simunic, Luca Benini, Peter Glynn, and Giovanni De Micheli. Dynamic power management for portable systems. In *Proceedings of the 6th annual international conference on Mobile computing and networking - MobiCom '00*, pages 11–19, New York, New York, USA, 2000. ACM Press. ISBN 1581131976.

- doi: 10.1145/345910.345914. URL <http://portal.acm.org/citation.cfm?doid=345910.345914>.
- [20] Ying Tan, Wei Liu, and Qinru Qiu. Adaptive power management using reinforcement learning. *Proceedings of the 2009 International Conference on Computer-Aided Design - ICCAD '09*, page 461, 2009. doi: 10.1145/1687399.1687486. URL <http://portal.acm.org/citation.cfm?doid=1687399.1687486>.
- [21] Hung-Cheng Shih and Kuochen Wang. An adaptive hybrid dynamic power management algorithm for mobile devices. *Computer Networks*, 56(2):548–565, February 2012. ISSN 13891286. doi: 10.1016/j.comnet.2011.10.005. URL <http://linkinghub.elsevier.com/retrieve/pii/S1389128611003719>.
- [22] Gaurav Dhiman and T.S. Rosing. System-level power management using online learning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(5):676–689, May 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2015740. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4838819>http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4838819.
- [23] Young-si Hwang, Sung-kwan Ku, and Ki-seok Chung. A predictive dynamic power management technique for embedded mobile devices. *IEEE Transactions on Consumer Electronics*, 56(2):713–719, May 2010. ISSN 0098-3063. doi: 10.1109/TCE.2010.5505992. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5505992>.
- [24] Viswanathan Lakshmi Prabha and Elwin Chandra Monie. Hardware Architecture of Reinforcement Learning Scheme for Dynamic Power Management in Embedded Systems. *EURASIP Journal on Embedded Systems*, 2007:1–6, 2007. ISSN 1687-3955. doi: 10.1155/2007/65478. URL <http://www.hindawi.com/journals/es/2007/065478/abs/>.
- [25] Wei Liu, Ying Tan, and Qinru Qiu. Enhanced Q-learning algorithm for dynamic power management with performance constraint. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 602–605. European Design and Automation Association, 2010. ISBN 9783981080162. URL <http://dl.acm.org/citation.cfm?id=1871068>.
- [26] C. Gniady, A.R. Butt, and Y.C. Hu. Program counter-based prediction techniques for dynamic power management. *IEEE Transactions on Computers*, 55(6):641–658, June 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.87. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1628954>.
- [27] Khurram Bhatti, Cecile Belleudy, and Michel Auguin. Power Management in Real Time Embedded Systems through Online and Adaptive Interplay of DPM

- and DVFS Policies. *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 184–191, December 2010. doi: 10.1109/EUC.2010.35. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5703515>.
- [28] Jie Chen, Deyuan Gao, and Qiaoshi Zheng. A research on an optimized adaptive dynamic power management. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 52–55. Ieee, 2009. ISBN 978-1-4244-4519-6. doi: 10.1109/ICCSIT.2009.5234610. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5234610>.
- [29] Yang Ge and Qinru Qiu. Dynamic thermal management for multimedia applications using machine learning. In *Proceedings of the 48th Design Automation Conference on - DAC '11*, page 95, New York, New York, USA, 2011. ACM Press. ISBN 9781450306362. doi: 10.1145/2024724.2024746. URL <http://dl.acm.org/citation.cfm?doid=2024724.2024746>.
- [30] Advanced Configuration and Power Interface, 2015. URL http://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf.
- [31] Amit Kumar Singh, Anup Das, and Akash Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 115:1–115:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488875. URL <http://doi.acm.org/10.1145/2463209.2488875>.
- [32] Anup Das, Geoff V Merrett, and Bashir M Al-hashimi. The Slowdown or Race-to-idle Question : Workload-Aware Energy Optimization of SMT Multicore Platforms under Process Variation. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 535–538, 2016.
- [33] H. Posadas, E. Villar, D. Ragot, and M. Martinez. Early modeling of linux-based rtos platforms in a systemc time-approximate co-simulation environment. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 238–244, May 2010. doi: 10.1109/ISORC.2010.18.
- [34] Kihwan Choi, Wei-Chung Cheng, and Massoud Pedram. Frame-Based Dynamic Voltage and Frequency Scaling for an MPEG Player. *Journal of Low Power Electronics*, 1(1):27–43, April 2005. ISSN 15461998. doi: 10.1166/jolpe.2005.005. URL <http://openurl.ingenta.com/content/xref?genre=article&issn=1546-1998&volume=1&issue=1&page=27>.
- [35] Sung-yong Bang, Kwanhu Bang, Sungroh Yoon, and Eui-young Chung. Run-Time Adaptive Workload Estimation for Dynamic Voltage Scaling. *IEEE Transactions*

- on Computer-Aided Design of Integrated Circuits and Systems*, 28(9):1334–1347, September 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2024706. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5208582>.
- [36] Yan Gu and Samarjit Chakraborty. Control theory-based DVS for interactive 3D games. In *Proceedings - Design Automation Conference*, pages 740–745, New York, New York, USA, 2008. ACM Press. ISBN 9781605581156. doi: 10.1109/DAC.2008.4555917. URL <http://portal.acm.org/citation.cfm?doid=1391469.1391659>.
- [37] Candy Pang, Abram Hindle, Bram Adams, and Ahmed Hassan. What do programmers know about software energy consumption? *IEEE Software*, pages 1–1, 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.83. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7155416>.
- [38] Brian Carlson and Bill Giolma. SmartReflex Power and Performance Management Technologies : reduced power consumption, optimized performance. 2008. URL http://focus.ti.com/pdfs/wtbu/smartreflex_whitepaper.pdf.
- [39] Luis Alfonso Maeda-Nunez, Anup Das, Rishad A Shafik, Geoff V Merrett, and Bashir M Al-Hashimi. PoGo : An Application-Specific Adaptive Energy Minimisation Approach for Embedded Systems. In *EEHCO*, pages 1–6, 2014.
- [40] Asieh Salehi Fathabadi, Luis Alfonso Maeda-Nunez, Michael Butler, Bashir Al-Hashimi, and Geoff Merrett. Towards automatic code generation of run-time power management for embedded systems using formal methods. In *9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC-15)*, September 2015. URL <http://eprints.soton.ac.uk/379127/>.
- [41] Rishad Ahmed Shafik, Anup K. Das, Luis Alfonso Maeda-Nunez, Sheng Yang, Geoff V. Merrett, and Bashir Al-Hashimi. Learning transfer-based adaptive energy minimization in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–14, August 2015. URL <http://eprints.soton.ac.uk/374893/>.
- [42] Sascha Bischoff, Andreas Hansson, and Bashir M. Al-Hashimi. Applying of Quality of Experience to system optimisation. In *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 91–98. Ieee, September 2013. ISBN 978-1-4799-1170-7. doi: 10.1109/PATMOS.2013.6662160. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6662160>.
- [43] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual For System-on-Chip Design*. Springer, 2007.
- [44] Low-power flow user guide user guide, December 2010.

- [45] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of the 2007 international symposium on Low power electronics and design - ISLPED '07*, pages 207–212, New York, New York, USA, 2007. ACM Press. ISBN 9781595937094. doi: 10.1145/1283780.1283825. URL <http://dl.acm.org/citation.cfm?id=1283825><http://portal.acm.org/citation.cfm?doid=1283780.1283825>.
- [46] Masakatsu Nakai, Satoshi Akui, Katsunori Seno, Tetsumasa Meguro, Takahiro Seki, Tetsuo Kondo, A. Hashiguchi, H. Kawahara, K. Kumano, and M. Shimura. Dynamic voltage and frequency management for a low-power embedded microprocessor. *IEEE Journal of Solid-State Circuits*, 40(1):28–35, January 2005. ISSN 0018-9200. doi: 10.1109/JSSC.2004.838021. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1374987<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1374987>.
- [47] Texas Instruments. AM-DM37x Technical Reference, 2012. URL <http://www.ti.com/litv/pdf/sprugn4r>.
- [48] Robert Oshana. *DSP Software Development Techniques for Embedded and Real-Time Systems*. Elsevier Inc., 2006. ISBN 9780750677592.
- [49] Chi-Hong Hwang and Allen C.-H. Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):226–241, April 2000. ISSN 10844309. doi: 10.1145/335043.335046. URL <http://portal.acm.org/citation.cfm?doid=335043.335046>.
- [50] Ali Abbasian, Safar Hatami, A. Afzali-Kusha, Mehrdad Nourani, and Caro Lucas. Event-driven dynamic power management based on wavelet forecasting theory. In *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, number c, pages V–325–V–328. IEEE, 2004. ISBN 0-7803-8251-X. doi: 10.1109/ISCAS.2004.1329528. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1329528>.
- [51] Gaurav Dhiman and Tajana Rosing. Dynamic Power Management Using Machine Learning. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 747–754. IEEE, November 2006. ISBN 1-59593-389-1. doi: 10.1109/ICCAD.2006.320115. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4110117>.
- [52] Nevine AbouGhazaleh, Alexandre Ferreira, Cosmin Rusu, Ruibin Xu, Frank Liberato, Bruce Childers, Daniel Mosse, and Rami Melhem. Integrated CPU and l2 cache voltage scaling using machine learning. *ACM SIGPLAN Notices*, 42(7):41, July 2007. ISSN 03621340. doi: 10.1145/

- 1273444.1254773. URL <http://portal.acm.org/citation.cfm?doid=1273444.1254773><http://dl.acm.org/citation.cfm?id=1254773>.
- [53] Saurabh Sinha, Jounghyuk Suh, Bertan Bakkaloglu, and Yu Cao. Workload-Aware Neuromorphic Design of the Power Controller. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(3):381–390, September 2011. ISSN 2156-3357. doi: 10.1109/JETCAS.2011.2165233. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6035994>.
- [54] Saurabh Sinha, Jounghyuk Suh, Bertan Bakkaloglu, and Yu Cao. Workload-aware neuromorphic design of low-power supply voltage controller. *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design - ISLPED '10*, 1:241, 2010. doi: 10.1145/1840845.1840896. URL <http://portal.acm.org/citation.cfm?doid=1840845.1840896>.
- [55] Saurabh Sinha, Jounghyuk Suh, Bertan Bakkaloglu, and Yu Cao. A workload-aware neuromorphic controller for dynamic power and thermal management. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 200–207. Ieee, June 2011. ISBN 978-1-4577-0598-4. doi: 10.1109/AHS.2011.5963936. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5963936>.
- [56] Amit Sinha and A.P. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 221–226. IEEE Comput. Soc, 2001. ISBN 0-7695-0831-6. doi: 10.1109/ICVD.2001.902664. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=902664<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=902664>.
- [57] M. Najibi, M Salehi, A. Kusha, M. Pedram, S. Fakhraie, and H. Pedram. Dynamic Voltage and Frequency Management Based on Variable Update Intervals for Frequency Setting. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 755–760. IEEE, November 2006. ISBN 1-59593-389-1. doi: 10.1109/ICCAD.2006.320116. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4110118>.
- [58] Mostafa E. Salehi, Mehrzad Samadi, Mehrdad Najibi, Ali Afzali-Kusha, Masoud Pedram, and Sied Mehdi Fakhraie. Dynamic Voltage and Frequency Scheduling for Embedded Processors Considering Power/Performance Tradeoffs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10):1931–1935, October 2011. ISSN 1063-8210. doi: 10.1109/TVLSI.2010.2057520. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5545490>.
- [59] Wolfgang Maurer. *Professional Linux kernel architecture*. Wiley Publishing, Inc., 2010. ISBN 9780470343432. URL <http://books.google.com/books?hl=>

en&lr=&id=-6zvRFEfQ24C&oi=fnd&pg=PT9&dq=Professional+Linux+Kernel+Architecture&ots=8DnACkWaUL&sig=6r3rAz0gTN5kZ36E10gUaUDv2G4.

- [60] Greg Kroah-Hartman. *LINUX KERNEL in a Nutshell*. O'REILLY, 2006.
- [61] Peter Jay Salzman. *The Linux Kernel Module Programming Guide*. 2007.
- [62] Muhammad Khurram Bhatti, Cécile Belleudy, and Michel Auguin. Hybrid power management in real time embedded systems: An interplay of DVFS and DPM techniques. *Real-Time Systems*, 47(2):143–162, January 2011. ISSN 09226443. doi: 10.1007/s11241-011-9116-y. URL <http://www.springerlink.com/index/10.1007/s11241-011-9116-y>.
- [63] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 28. Cambridge Univ Press, 1998. URL <http://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=34656>.
- [64] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988. ISSN 0885-6125. doi: 10.1007/BF00115009. URL <http://www.springerlink.com/index/10.1007/BF00115009>.
- [65] C.H.C. Ribeiro. A tutorial on reinforcement learning techniques. In *Supervised Learning track tutorials of the 1999 International Joint Conference on Neural Networks*, 1999. URL http://www.ppgia.pucpr.br/~fabricio/ftp/PIBIC/Artigos/Ag_Aprendizagem/TAIC-tutorial_RL.pdf.
- [66] Csaba Szepesvári. *Algorithms for reinforcement learning*. 2010. URL <http://www.morganclaypool.com/doi/pdf/10.2200/S00268ED1V01Y201005AIM009>.
- [67] G. E. P. Box. Understanding Exponential Smoothing – A Simple Way to Forecast Sales and Inventory. *Quality Engineering*, 1990.
- [68] Harriet Black Nembhard and Ming Shu Kao. Adaptive Forecast-Based Monitoring for Dynamic Systems. *Technometrics*, 45(3):208–219, August 2003. ISSN 0040-1706. doi: 10.1198/004017003000000032.
- [69] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'REILLY, 3rd edition, 2005. ISBN 0596555385. URL <http://books.google.com/books?id=M7RHMAcEk4C&pgis=1>.
- [70] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'REILLY, 3 edition, 2005.
- [71] Agner Fog. Optimizing software in C++, 2015. URL http://www.agner.org/optimize/optimizing_cpp.pdf.
- [72] Agner Fog. Optimizing subroutines in assembly language, 2015. URL http://www.agner.org/optimize/optimizing_assembly.pdf.

- [73] ARM. ARM Cortex-A8 Reference Manual, 2010.
- [74] Arnaldo Carvalho de Melo. The new linuxperf tools. 2010.
- [75] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.
- [76] Heike McCraw, Joseph Ralph, Anthony Danalis, and Jack Dongarra. Power monitoring with papi for extreme scale architectures and dataflow-based programming models. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 385–391. IEEE, 2014.
- [77] perfmon2: improving performance monitoring on Linux, 2008. URL <http://perfmon2.sourceforge.net/>.
- [78] Arm Limited. Cortex -A9 Technical Reference Manual, 2010.
- [79] Robert C. Nelson. BeagleBoardUbuntu, 2013. URL <http://elinux.org/BeagleBoardUbuntu>.
- [80] FFMPEG. URL Available:<http://www.ffmpeg.org>.

Appendix A - Sample Shepherd Output Files

The following is a sample output file autogenerated to track the performance of Shepherd. It logs the performance of every frame for the application. The log includes the frame number, the type, completion time (microseconds), whether the deadline was met or not, the predicted workload for that frame, the real workload for that frame and the frequency at which said frame was run. This Comma-Separated Values (CSV) file is generated at start of the application and it is populated by writing to it every frame.

```
VIDEO DATA. PID:4092. FILE: videos_paper//h264/vga/video5_h264_vga.mp4
FRAME_NUM,FRAME_TYPE,FRAME_TYPE_NUM,FRAME_TIME_MICROSECONDS,DEADLINE_PASSED,
    PREDICTED_WORKLOAD,ACTUAL_WORKLOAD,FREQUENCY
1,I,1,0,1,0,0,300000
2,I,1,0,1,0,5433316,600000
3,P,2,0,0,3259984,30343512,300000
4,B,3,31708,1,0,116255172,600000
5,B,3,23254,1,0,22489112,1000000
6,B,3,21026,1,13493456,41029912,1000000
7,P,2,69061,0,30015312,41796068,300000
8,B,3,26398,1,69753088,23360374,1000000
9,B,3,70557,0,37083760,31525938,300000
10,B,3,26581,1,33749056,23846062,1000000
11,P,2,27679,1,27807248,31855986,1000000
12,B,3,35522,1,41917456,33261682,600000
13,B,3,29114,1,30236480,24830506,800000
14,B,3,33997,1,26992880,27716538,600000
15,P,2,32959,1,27427056,24017628,800000
16,B,3,59387,0,36723984,30777774,300000
17,B,3,25726,1,25381392,20356236,1000000
18,B,3,34576,1,22366288,30800898,600000
19,P,2,37048,1,27427040,24460908,800000
20,B,3,65002,0,33156240,33990222,300000
21,B,3,31494,1,25647344,22055206,800000
22,B,3,37170,1,23492048,29392808,600000
23,P,2,36225,1,27032496,25958090,800000
24,B,3,36865,1,33656608,33173566,600000
25,B,3,36927,1,26387840,25767028,600000
26,P,2,45960,0,26015344,25742000,600000
27,B,3,37781,1,33366768,31294992,600000
28,B,3,37780,1,25851328,26350620,600000
29,B,3,35096,1,26150896,26459676,600000
```

```
30,P,2,34546,1,26336144,24790768,800000
31,B,3,62226,0,32123696,32065024,300000
32,B,3,62561,0,25408912,21212506,300000
33,B,3,33905,1,22891056,21322526,600000
34,P,2,35400,1,21949920,23921540,800000
35,B,3,34973,1,32088480,32450820,600000
36,B,3,36102,1,23132880,24501138,600000
37,B,3,29236,1,23953824,25309106,800000
38,P,2,35736,1,24766992,27563464,800000
39,B,3,37354,1,32305872,33369426,600000
40,B,3,67841,0,26444864,25916356,300000
41,B,3,36926,1,26127744,22912446,600000
42,P,2,30395,1,24198544,25623562,800000
43,P,2,41565,1,32944000,28683138,600000
44,B,3,36102,1,30387472,28492234,600000
45,B,3,35675,1,25053536,25350366,600000
46,B,3,33295,1,25231616,25088334,600000
47,P,2,29846,1,25145632,23632122,800000
48,B,3,35035,1,29250320,28179250,600000
49,B,3,61340,0,24237520,24634166,300000
50,B,3,33539,1,24475504,20947184,600000
51,I,1,138337,0,22358512,25332212,300000
52,P,2,38788,1,19510096,44081364,600000
.
.
.
690,B,3,26947,1,40763968,41778218,1000000
691,B,3,28137,1,41559568,41578226,1000000
692,B,3,26794,1,41570752,41623446,1000000
693,P,2,31005,1,41602352,42026928,1000000
694,B,3,27985,1,41372512,41432444,1000000
695,B,3,28107,1,41857088,41821292,1000000
696,B,3,26306,1,41835600,42107340,1000000
697,P,2,30243,1,41998624,41234106,1000000
698,B,3,27618,1,41408464,42026756,1000000
699,B,3,28686,1,41539904,41481830,1000000
700,B,3,26825,1,41505056,41422440,1000000
701,P,2,27221,1,41455472,41912570,1000000
702,B,3,27557,1,41779424,41742816,1000000
703,B,3,27130,1,41729712,41572410,1000000
704,I,1,55694,0,41635312,42755964,800000
705,P,2,27741,1,42532640,49156684,1000000
706,B,3,24658,1,41757456,33043368,1000000
707,B,3,25086,1,42307696,30209394,1000000
708,P,2,33661,1,35048704,40929476,800000
709,B,3,27954,1,36528992,33230516,800000
710,B,3,29633,1,38577152,33492902,800000
711,P,2,27618,1,35526592,33529458,1000000
712,B,3,24750,1,34549904,41123950,1000000
713,P,2,34332,1,34328304,41862642,800000
714,B,3,31097,1,38494320,33612010,800000
715,P,2,33143,1,38848896,32946288,800000
716,B,3,28412,1,35564928,33635596,800000
717,P,2,30914,1,35307328,33337320,800000
718,B,3,29664,1,34407312,33045544,800000
719,P,2,31250,1,34125312,33647502,800000
720,B,3,30060,1,33590240,33442002,800000
721,P,2,29327,1,33838608,33353570,800000
```

Appendix B - Publications

The following Appendix presents the publications generated with this research.

Three publications have been generated:

- “PoGo : An Application-Specific Adaptive Energy Minimisation Approach for Embedded Systems” published in HiPEAC 2015 [39]
- “Towards automatic code generation of run-time power management for embedded systems using formal methods” published in MCSoc 2015 [40]
- “Learning transfer-based adaptive energy minimization in embedded systems” published at IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, August 2015 [41]

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).

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

```

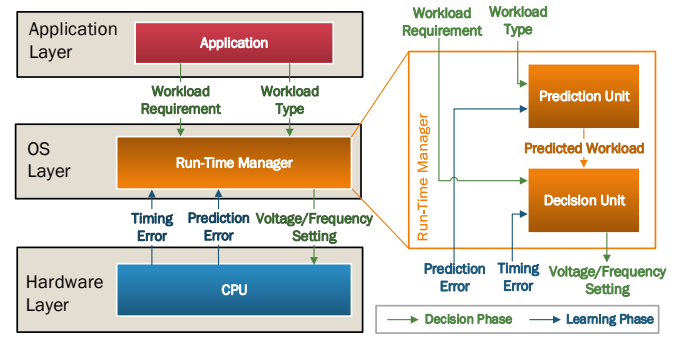


Figure 1. Run-Time Management Unit in the cross-layer approach

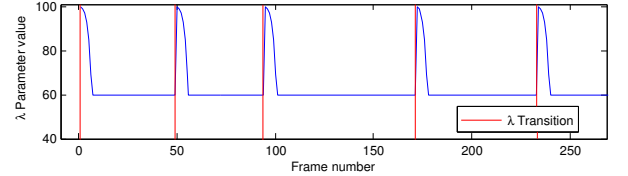


Figure 2. AEWMA λ parameter change at transitions

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:

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

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:

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

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 type¹ 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,

¹For 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 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 epoch², 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 decisions³. 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.

²In reinforcement learning terminology, the interval at which the algorithm is triggered is known as decision epoch.

³Appropriate decisions are those that reduce the energy consumption, while satisfying the performance.

A) Exploration phase						B) Exploitation phase					
		ACTIONS (V-F pairs)						ACTIONS (V-F pairs)			
		V1,F1	V2,F2	V3,F3	V4,F4			V1,F1	V2,F2	V3,F3	V4,F4
STATES (Workload Amount)	0	0.0	0.0	0.0	0.0	0	0.8	0.2	0.2	0.1	
	1	0.0	0.0	0.0	0.0	1	-0.1	0.6	0.4	0.2	
	2	0.0	0.0	0.0	0.0	2	-0.4	-0.2	0.7	0.3	
	3	0.0	0.0	0.0	0.0	3	-0.7	-0.3	0.8	0.2	
	4	0.0	0.0	0.0	0.0	4	-1.0	-0.8	-0.1	0.8	

Several epochs later

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.

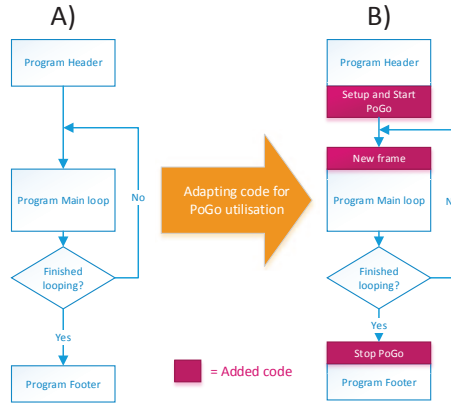


Figure 4. Adaptation of program code for utilisation of the PoGo Run-Time Manager

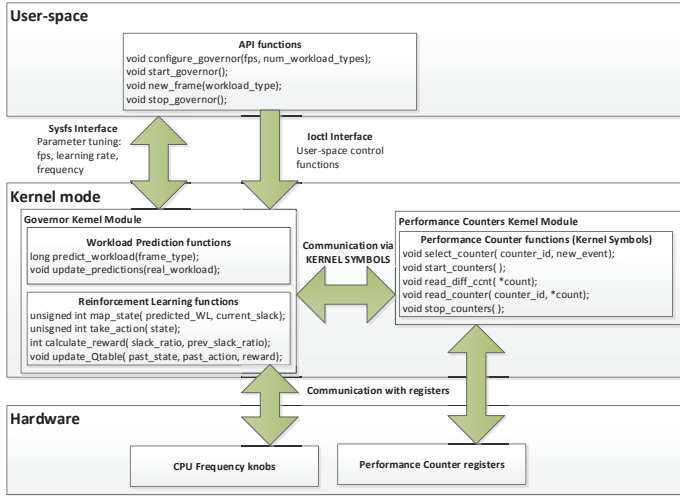


Figure 5. PoGo governor implementation

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^t . 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.

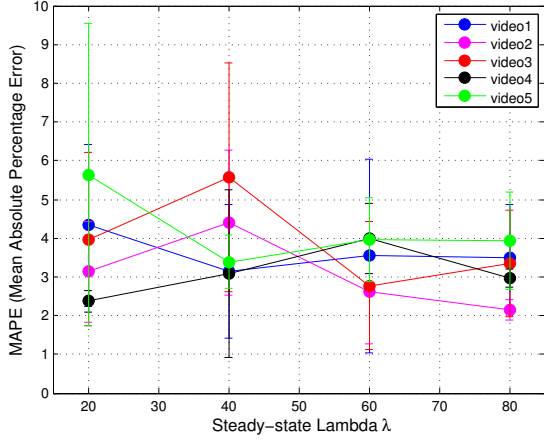


Figure 6. Effect of steady-state weight of λ for AEWMA on prediction error using dynamic workloads

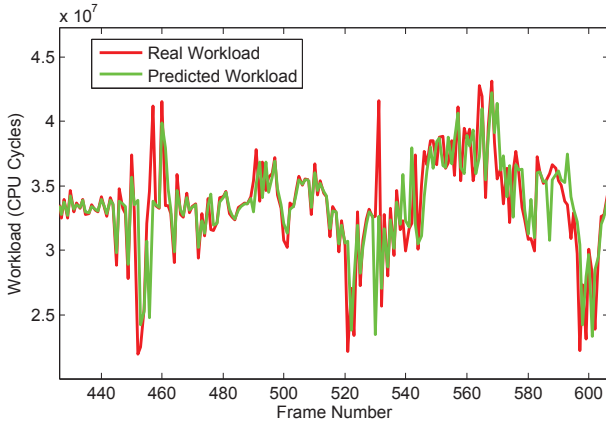


Figure 7. Comparison of real workload vs. predicted

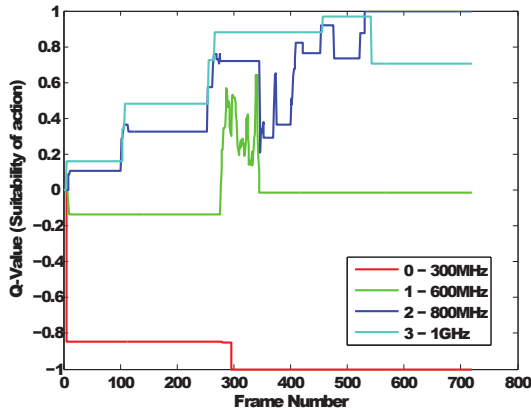


Figure 8. Q-values for different actions in state 4

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).

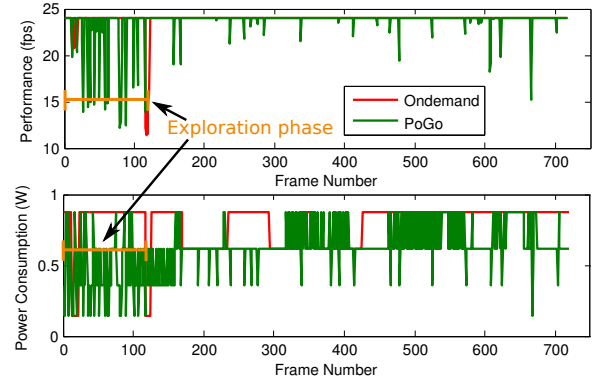


Figure 9. Performance and Power Consumption of the Governors PoGo and Ondemand for an H.264 Video

- **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.

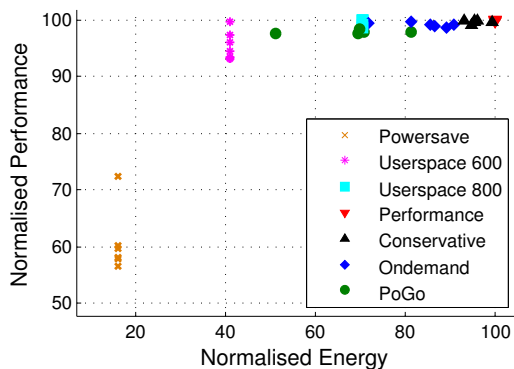


Figure 10. Pareto graph comparing different Governors vs. PoGo by means of energy and performance for H.264 workloads

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 μ s to compute (from the time of the system call to the end of the

Governor	Objective: 8fps		Objective: 10fps	
	Performance (fps)	Normalised Energy	Performance (fps)	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.

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.

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.

Towards Automatic Code Generation of Run-Time Power Management for Embedded Systems using Formal Methods

Asieh Salehi Fathabadi, Luis Alfonso Maeda-Nunez, Michael J Butler, Bashir M Al-Hashimi, and Geoff V Merrett
Electronics and Computer Science
University of Southampton
{asf08r, lm15g10, mjb, bmah, gvm}@ecs.soton.ac.uk

Abstract—Run-Time Management (RTM) systems are used to control energy hooks at run-time to minimise the energy consumption of embedded systems with single and many-core processors. Typically, such RTM systems are aware of application requirements and utilise workload prediction and machine learning algorithms to estimate the optimal configuration. An RTM mechanism should not compromise the reliability or performance of the platform it is managing. Because of the potential complexity and interaction with the platform and its applications, we are using rigorous design methods that allow us to master the complexity and verify the correctness of our designs in a formal way. The formal RTM design can be verified earlier in the development process before implementation, which early verification can reduce the cost of fixing potential failures which can be very demanding in testing the system after implementation. In addition, the formal model of a RTM system can be automatically translated into executable code to be executed on the hardware. Automatic code generation reduces the efforts of hand-coded implementation and is portable across different architectures and Operating Systems (OSs). In this paper we propose a formal approach toward automatic generation of RTM system code, for a video decoder application, from a verified formal model of a RTM. The formal model of the RTM system is developed using the Event-B formal modelling language and is verified using theorem proving and model checking. The automatically generated RTM system has been integrated in an embedded platform as a Linux governor, and provides up to 4% improvement over Linux's default Ondemand governor.

I. INTRODUCTION

Dynamic Voltage and Frequency Scaling (DVFS) has been widely used to reduce the energy consumption of mobile and embedded systems at run-time, while maintaining a required Quality of Service (QoS) [1–5]. To manage DVFS, a Run-Time Management (RTM) system is an essential unit in a many-core architecture, and it needs to interact with both the application layer (to ensure that QoS requirements are met) and the hardware layer (to control and monitor core activities). In addition, the RTM typically includes workload prediction and machine learning algorithms. Therefore, ensuring an integrated and reliable RTM system is not trivial to achieve using a manual hand-coded implementation. Hand-coded RTM system implementation can be error-prone, and is not portable across different architectures and Operating Systems (OSs).

In this paper, we address the integrity and reliability of the RTM through deployment of formal design and verification methods. Once a system is modelled mathematically, proof

can be used to ensure the correctness and consistency of the model. We use the Event-B formal method [6] to model and verify a RTM system. The use of formal methods [7] helps to reduce costs by identifying specification and design errors at early development stages before implementation when they are cheaper to fix. The verified Event-B model of the RTM system can be translated into executable code which can be executed on the hardware. Moreover, representing runtime algorithms more abstractly allows us to target different architectures and OS through code generation. This has the potential for future savings allowing the same algorithm to be portable across different architectures and OS by tailoring the code generation.

This paper reports on our initial experimentation of automatic code generation of a RTM system from Event-B models. The RTM system performs DVFS for a video decoder application. The RTM system is modelled by Event-B, and is verified by theorem proving and model checking techniques. The main contribution of this work is experimenting with the automatic generation of RTM code through a reliable verification approach. While the current work has considered a single-core platform for a proof of concept, modelling and verifying a RTM for many-core hardware is considered as a future work.

The paper starts with related works in Section II followed by description of RTM in Section III. Then formal design and the Event-B model are presented in Section IV. Section V is about verifying the formal models. The implementation including the code generation technique is explained in Section VI. Finally the experimental results are shown in Section VII followed by the conclusion section.

II. RELATED WORKS

Power management hardware has been designed and implemented in embedded systems to provide energy savings and temperature stability, with Dynamic Power Management (DPM) and DVFS being two techniques controllable from software. The hardware implementation of these techniques has been described by Keating et al. [8] (DPM being referred to as Power Gating). Power management mechanisms for control of DPM and DVFS have been widely studied in the literature. For the case of DVFS power management, studies can be divided into performance requirement-agnostic and performance requirement-aware. The first are focused on optimising the energy consumption without knowledge of the

application requirements. The *Ondemand* governor [9] is a DVFS controller in Linux that reacts to current processor workload, adjusting Voltage and Frequency (V-F) to maintain an idle time setting. The work of Dhiman et al. [10, 11] presents a control algorithm that characterises workload based on the performance monitors, adjusting the V-F according to an energy-performance trade-off. The works of [12, 13] use DVFS by predicting the workload of a time period, aiming to reduce idle time by changing the V-F. Multi-core approaches for performance requirement-agnostic DVFS include [5, 14]. Moeng et al. [5] do offline workload characterisation based on performance counters, producing a decision tree for run-time for optimising the energy per user instruction. Bose et al. [14] provide a summary of multi-core DVFS, with further guidelines for designing system-level RTM.

Performance requirement-aware studies reduce energy consumption whilst aiming to preserve a performance requirement, either provided by the application or the scheduler. Real-time systems have this requirement, as their real-time tasks are executed to meet their intrinsic deadline. The work of Bhatti [4] focuses on RTM for real-time system. Soft real-time systems have this performance requirement but with a less strict directive, thus it is not critical if the deadline is not met. Video decoding applications present this soft real-time property, as the requirement is set by the frames-per-second (FPS) required by the video, and missing the FPS is not system critical. Work from [1–3, 15] focuses on these applications, using workload prediction to determine the V-F setting before the frame is computed. As a starting point for the development of the RTM Event-B model in this paper, the algorithm is focused on performance requirement-aware applications, namely video decoding. The algorithm is based on the work of [15], which uses prediction for estimating the workload, and reinforcement learning to select the V-F setting.

Salehi et al. [16] present several Event-B modelling and verification techniques used in the development of the RTM of the video decoder system. While this includes more details about the application of different modelling/verification techniques, in this paper we present more detailed results about the automatic generation of the RTM implementation code from an Event-B model. Code generation technique has been introduced in the Event-B formal method to bridge the gap between abstract specifications and implementation using an implementation level specification notion [17].

III. RUN-TIME MANAGEMENT (RTM)

The design of the RTM is described in this section, which involves workload characterization together with the appropriate V-F selection. Figure 1 shows the RTM adopting a cross-layer approach, interacting with the application, OS and hardware. Communication between layers is indicated by arrows.

The objective of real-time applications is to complete the execution of their workload before a predefined deadline. In hard real-time systems, both the workload and the deadline of the task must be known before starting processing it, so that the scheduler can allocate the task for execution. In soft real-time systems, tasks may or may not provide this information, so it needs to be calculated if performance (or power) is to be optimised. Because scheduling cannot be deterministic,

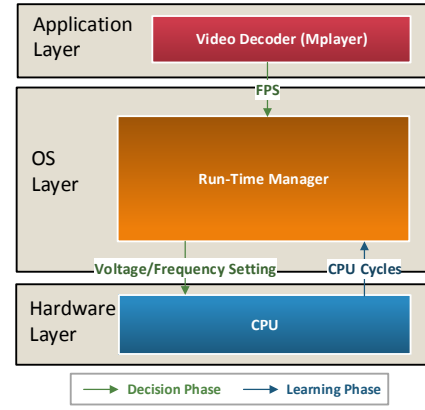


Fig. 1. Run-Time Management system for a video decoder application: a cross-layer approach

the completion of the task before the deadline cannot be guaranteed. Video decoders present this behaviour as the frame can be seen as a task, and the workload information per frame may not be known before its decoding. The deadline can be obtained from the FPS set by the video application. The power minimization objective for these applications then translates in the solution to a constrained optimization problem. The frame workload needs to be known (to a certain extent) prior to its processing, then decisions on the power state (V-F) have to be taken so that they fulfil the constraint but take into account performance variations of the application.

To achieve this soft real-time behaviour, our RTM algorithm is based on [15] and works in two phases, *Prediction* and *Decision Making*. For each frame, the RTM first predicts the workload to be executed, and then it decides the V-F setting so that the predicted workload can finish execution before the frame deadline, set by the FPS. After the frame has been executed, the RTM learns by using feedback for updating its parameters for computing future frames. To achieve the first objective, predictions of the workload for the next frame are performed using an Exponential Weighted Moving Average (EWMA)[18]. The EWMA algorithm is explained in Section IV-D1. The work of [1, 12, 13] use EWMA for their workload prediction schemes, as it is easy to implement, lightweight at runtime and presents good performance. For the Decision Making, Reinforcement Learning (RL) is used [19], using the Q-Learning algorithm. The methodology to design the algorithm using formal methods is explained next.

IV. FORMAL DESIGN

A. Formal Methods and Event-B

In computer science, *formal methods* [7] are mathematically based techniques for the specification and development of complex systems. By building a mathematically rigorous model of a system, it is possible to verify the system's properties in a more thorough fashion than empirical testing, and therefore it improves reliability and robustness of design. *Event-B* [6] is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory and first order logic as a modelling notation, the use of refinement to represent systems at different abstraction levels, and the

use of mathematical proof to verify correctness of models and consistency between refinement levels. The behaviour of an Event-B model is defined by a collection of variables together with a collection of guarded atomic events that modify the variables. Formal properties are specified using invariants and preservation of invariants by events is verified using proof techniques.

The *Rodin* [20] platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof.

B. Formal Design Architecture

Figure 2 illustrates our design architecture for Event-B modelling of the RTM for a video decoder system. Event-B refinement allows a model to be built gradually, starting with an abstract model and then introducing successive, more concrete refinements. As shown in Figure 2, the Event-B model of the RTM system comprises an abstraction level, where we focus on the main functionalities of the system, and two levels of refinements, where the workload prediction and the reinforcement learning algorithms are introduced respectively. To manage the complexity of the final refinement and also to prepare the model for code generation, the model is decomposed into two sub-models: Controller and Environment. The Controller sub-model consists of properties of the RTM algorithms and the environment sub-model represents the interfaces between the RTM and the application and hardware layers (see Figure 1). By separating controller behaviour and environment behaviour, the representation of the RTM layer and the application and hardware layers are divided. This structure is used for code generation configuration, where the controller translation consists of RTM algorithms, and environment translation represents the interfaces to the application and hardware layers. Details of this implementation are explained in Section VI. The sub-models need some preparation before the final step of being translated to the executable code. These preparations are included in refinement levels of sub-models: Controller tasking refinement and Environment tasking refinement. In these refinements, the control flows, sequencing and branching of Event-B actions are defined. Also additional translation rules are defined before attempting code generation. These translation rules specify the translation of the Event-B mathematical operators to corresponding C operators. Finally the C code is generated automatically from the final refinement models of the controller and the environment.

C. Abstraction

To present details of the Event-B abstract/refinement levels, we benefit from a visualisation approach, called Event Refinement Structures (ERS) [21]. The ERS of the RTM system is presented in Figure 3. The blue region shows the abstract level including four actions. Each node indicates an action in the Event-B model and the oval is the name of the system. The nodes are read from left to right indicating the ordering between them. First the *set_fps* event executes followed by execution of *select_vf*, *execute_frame* and *monitor_actwl*. According to the specification of the system described in Section III, first the value of FPS is provided by the application layer and saved in the RTM, then the optimal value of V-F is decided by the RTM, the frame is executed in the hardware and the actual value of workload *actwl* is monitored.

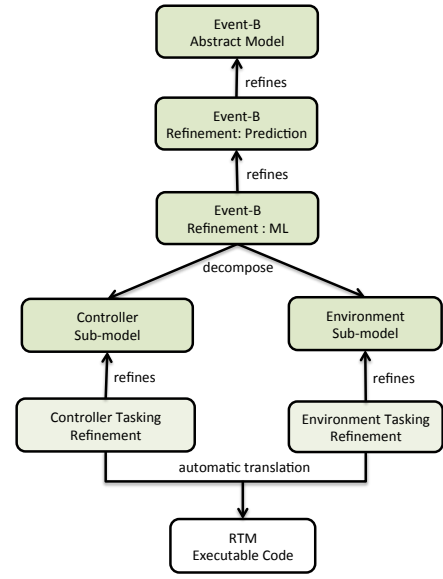


Fig. 2. Event-B formal design of the Run-Time Management system for a video decoder application

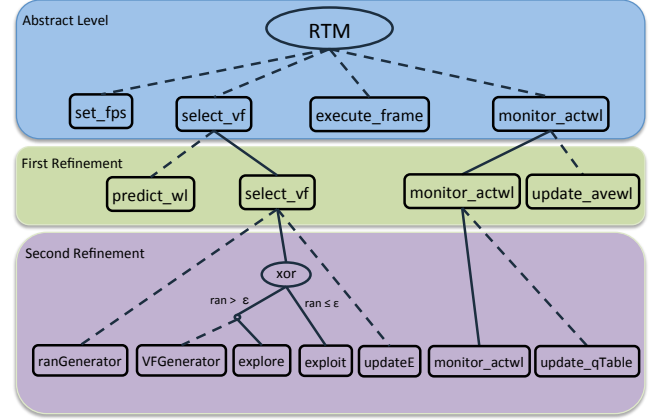


Fig. 3. Event refinement structure of the Run-Time Management system for a video decoder application

In the top level, the value of the V-F is decided non-deterministically from the constant set *VF*. Below is the Event-B specification of *select_vf* event. *act1* (action1) indicates the body of the event where the value of *VF* is non-deterministically assigned to a value from the set *VF*.

```

Event    select_vf ≐
begin
end      act1 : freq ∈ VF

```

D. First Refinement: Prediction Phase

1) *The Algorithm (EWMA)*: The prediction algorithm estimates the workload for the next frame using a modified form of EWMA. The EWMA algorithm is widely used in the literature [1, 12, 13] 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:

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

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:

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

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 [18].

2) *The Event-B Model*: In the abstract level, we do not model detailed workload prediction or the decision making. Later, in the first refinement (the green region of Figure 3), the details of the prediction algorithm are added to the abstract events: *select_vf* and *monitor_actwl*. The *select_vf* event is refined into two concrete events: *predict_wl*, where the workload is predicted and *select_vf*, where the value of V-F is decided based on our prediction. *monitor_actwl* event is also refined into two events: *monitor_actwl* (monitoring the actual workload) and *update_avgwl* (updating the average workload according to the equation 2).

In ERS, the line types indicated that whether the corresponding event is a refining event (solid line) or a new event (dashed line). In refining the *select_vf* event, *predict_wl* is a new event and the concrete *select_vf* event is refining the abstract *select_vf*. The description of these events are as below:

```

Event   predict_wl ≐
begin

end      act1 : pwl := predict(avgwl)

Event   select_vf ≐
refines select_vf
begin

end      act1 : freq := pwl * fps

```

In the *predict_wl* event, the *predict workload* variable (*pwl*) is assigned to the predicted value through the *predict* operator from the EWMA theory. A *theory* is an Event-B component where we can introduce new operators. In this development, we have defined a theory of EWMA where the prediction algorithm operators are defined. Later the value of *freq* is calculated based on the predicted workload in *select_vf* event.

E. Second Refinement: Decision Making

1) *The Algorithm (Reinforcement Learning)*: Once the workload for the future frame is predicted, the decision algorithm selects a V-F pair to execute it. This selection is based on the performance constraint in FPS given by the video application. The decision algorithm uses Q-Learning (Reinforcement

Learning). 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 is to learn how to make better decisions under variations. Decisions in reinforcement learning terminology are known as an actions, and the environment is represented as states. At the beginning there is no knowledge of the system, so the decision algorithm 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 epoch¹, 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. This reward/penalty is calculated with a cost function, which in this RTM context is defined as:

$$r = \begin{cases} \frac{100t}{d} & \text{if } d \leq t \\ -\frac{100(t-d)}{3d} & \text{if } t > d \end{cases} \quad (3)$$

where r is the reward, t is the workload time and d is the deadline. 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, 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 4 shows the evolution of the Q-Table. Initially, the values in the Q-Table are all zeros (Figure 4(A)); subsequently, in the exploitation phase, the ‘best’ actions are determined (in red in Figure 4(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 favour of appropriate decisions². 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.

¹In reinforcement learning terminology, the interval at which the algorithm is triggered is known as the decision epoch.

²Appropriate decisions are those that reduce the energy consumption, while satisfying the performance.

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	

Fig. 4. Q-Table during A) exploration and B) exploitation phases. The red boxes represent the best Action for each State.

2) *The Event-B Model*: The purple region (Figure 3) shows the second refinement, where details of reinforcement learning are included. The *select_vf* event is refined to include the details of the decision making algorithm. Based on comparing a random number, generated in *ranGenerator* event with the exploration-exploitation ratio (ϵ), either the *explore* or *exploit* event are executed and it is followed by updating the ϵ . The oval including *xor* presented an exclusive choice between its branches. Also the *monitor_actwl* event is refined to update the Q-Table: *update_qTable* event, where the workload is rewarded or penalized.

Below is the Event-B description of the *explore* and *exploit* events. These events are guarded based on the value of the *random* variable (generated in the *ranGenerator* event). If *random* is greater than the exploration-exploitation ratio (ϵ), *explore* executes, otherwise *exploit* executes. In the body of the *explore* event, the *freq* is assigned to a random value (was generated in the *VFGenerator*). The *explore* event assigns *freq* value into the optimal value of V-F according to the predicted workload (*pwl*). *optimalVF* is an operation defined in a theory where all of the necessary reinforcement learning operators are defined.

```

Event explore ≡
refines select_vf
when
begin grd : random > epsilon
end act1 : freq := randomVF

Event exploit ≡
refines select_vf
when
begin grd : random ≤ epsilon
end act1 : freq := optimalVF(QTable, pwl)

```

As shown in Figure 2, the final refinement is divided into two smaller sub-models. The controller sub-model includes the RTM actions: *predict_wl*, *ranGenerator*, *explore*, *exploit*, *VFGenerator*, *updateE* and *update_qTable*. The environment sub-model includes the actions to interact with the application and hardware: *set_fps*, *execute_frame* and *monitor_actwl*.

V. VERIFICATION

The correctness of an Event-B model is defined by invariant properties. An invariant is a predicate or constraint, which every state in the model must satisfy. More practically, every

event in the model must be shown to preserve this invariant. This verification requirement is expressed in a number of proof obligations (POs). In practice this verification is performed either by model checking or theorem proving (or both). In addition to correctness, the consistency of the refinement levels are proved by a number of proof obligations.

The Rodin toolset provides an environment for both theorem proving and model checking. PO generation, automatic proof and interactive proof are incorporated into Rodin. A user can prove a non-discharged proof obligation manually using the interactive proving feature of the Rodin.

Theorem proving There are different proof obligations which are generated by the Rodin, during development of a system [22]. The most important two of these are the Invariant Preservation (*INV*) proof obligation and the Guard Strengthening (*GRD*) proof obligation. The *INV* PO ensures that each invariant is preserved by each event; and the *GRD* PO ensures refinement consistency by ensuring that each abstract guard is no stronger than the concrete ones in the refining event. As a result, when a concrete event is enabled the corresponding abstract one is also enabled.

Model Checking ProB³ is an animator and model checker for Event-B. ProB allows fully automatic exploration of Event-B models, and can be used to systematically check a specification for range of errors.

The Event-B model of the RTM was verified using Rodin theorem proving. In the last refinement before model decomposition, 76 POs were generated, of which %96 are proved automatically. A manually proved PO is presented here as an example of verification.

The prediction refinement (Section IV-D) consist of two levels: in the first refinement an abstract representation of prediction (Section IV-D2) is modelled and in the second refinement it is proved that the abstract form is equivalent to the refining prediction formula presented in Equation (1). The abstract definition of prediction is defined in terms of the full history of measured workloads as follows:

$$pwl(n) = \lambda \sum_{i=1}^{n-1} actwlhst(i)(1-\lambda)^{n-i} + actwlhst(0)(1-\lambda)^n \quad (4)$$

Here *pwl*(*n*) is the predicted workload and *actwlhst*(*n*) is the actual work load (for the *n*th frame).

An invariant is defined in the refined model to specify that this abstract prediction is equivalent to Equation (1), i.e., the implementation of the algorithm is a correct refinement of the abstract prediction specification. This invariant is proved interactively with the Rodin theorem prover. Note that the abstract variable *actwlhst* (actual work load history) is not part of the refined model, it is only used for specification purposes.

We also analysed our model using ProB to ensure that the model is deadlock free. For each new event added in the refinements (events with dashed lines in Figure 3), we have verified that it would not introduce a deadlock using ProB. Also *INV* POs ensure that the new events keep the existing

³The ProB Animator and Model Checker: <http://www.stups.uni-duesseldorf.de/ProB/index.php5>

ordering constrains between the abstract events (ordering from left to right in Figure 3). The ordering between events are specified as invariants, the PO associated with each invariant ensures that its condition is preserved by each event.

VI. IMPLEMENTATION AND CODE GENERATION RESULTS

A. RTM Interface

The model of the RTM is automatically translated into C for its implementation. To provide genericity to the RTM model, the Controller sub-model does not take into account the hardware/application-specific calls needed to interact with the hardware and application layers (included in the Environment sub-model). Therefore, an interface to provide these functions has been designed. Figure 5 shows the modified RTM diagram from Figure 1, where the black box represents the generic RTM auto generated code, and the orange boxes provide the interactions with the hardware. The translated environmental functions are replaced by these interaction interfaces.

For this case study, in order for the generated RTM to sit at the OS layer, it has been implemented as a Linux Governor [9], which provides the interface and drivers to make the V-F changes. This Governor is composed of the three interfaces needed for the algorithm: the *Frequency Changer*, the *Performance Monitor* and the *Application Annotations*.

The *Frequency Changer* provides the *CPUFreq* drivers to change the V-F setting at the CPU. The *Performance Monitor* interface allows the system to recollect the CPU Cycles information from the hardware monitors. For the current case study, the architecture used is the ARM Cortex-A8, which provides Performance Monitoring registers [23]. A Loadable Kernel Module (LKM) was designed to access these monitors. The *Application Annotations* interface provides a library for the application (video decoder) to send its performance requirement (FPS) to the RTM. It also provides function calls to trigger the Governor to start and to finish working. It also notifies the RTM of a new frame start. This communication is done through *ioctl* calls. The interface uses an API [15] with the functions to be included in the application. After the RTM C code is generated, it is cross-compiled with the Governor interface to create the respective LKMs.

When the LKM is loaded, it waits for the *set_fps* (from the auto generated RTM) and the *start_governor* calls to start working. The *new_epoch* call at every new frame triggers the RTM algorithm for both learning (from previous frame) and deciding the new V-F. At the end of the application, the *stop_governor* call ends the RTM execution.

B. Code Generated RTM

According to Figure 2, after decomposition, the sub-models are refined to be prepared for translation into C code. Tasking Event-B sub-models define the control flows between events. Part of the controller tasking is as follows:

```
monitor_actwl;
update_avgwl;
if costFun_reward_assign
else costFun_penalty_assign;
```

It indicates the ordering between events *monitor_actwl* and *update_avgwl* followed by a branching between *costFun_reward_assign* event and *costFun_penalty_assign* event.

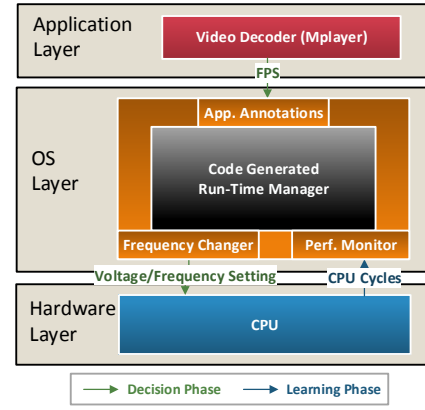


Fig. 5. Code Generated RTM for Video decoding

The later events are defined in the second refinement to represent the action of calculating the reward/penalty value (cost function) as part of updating the Q-Table.

The Event-B specification for *update_avgwl* is as follows:

```
Event   update_avgwl ≡
refines update_avgwl
begin
  act1 : avgwl := update(lambda, awl, avwl)
end
```

In the body of the event, action *act* updates the value of the variable *average workload*. The definition of *update* (according to Equation 1) is specified as an operator with three arguments as below:

```
Theory EWMA
operator update
arguments
  : lambda ∈ ℤ
  : avgWeight ∈ ℤ
  : nextWeight ∈ ℤ

Formula: : (lambda * nextWeight +
            (1 - lambda) * avgWeight)
```

The Event-B notation for the *costFun_reward_assign* is as follows:

```
Event   costFun_reward_assign ≡
refines costFun_reward_assign
when
  grd : (actwl / freq) ≤ dl
then
  act : costFun_reward_value :=
        min(1, costFun_reward(actwl, freq, dl))
```

The *costFun_reward_assign* can be executed only when its guard (*grd*) holds. *grd* condition specifies when the finish time is less than or equal to the deadline, means the deadline is achieved and the Q-Table needs to be rewarded. The *costFun_reward* is defined as an operator in the Machine Learning (ML) theory similar to what is described above for the *update* operation.

Below is part of the result of automatic code generation corresponding to the presented controller tasking part above:

```

Env_monitor_actwl(&actwl);
avgwl = (lambda * actwl + (1 - lambda) * avgwl);
if (actwl / freq <= dl) {
    costFun_reward_value = min(1,
        (100 * actwl) / (freq * dl));
} else {
    costFun_penalty_value = max(-1,
        -((actwl / freq - dl) * 100) / (3 * dl));
}

```

First the *monitor_actwl* is translated to a call to the environment function since *monitor_actwl* is an environment event. Then *update_avgwl* is translated into the second and third lines according to the operator definition for the *update*. Finally a branching is generated on the *costFun_reward_assign* and *costFun_penalty_assign* depending on the event guards. The Event-B guard of the event is translated into the branching condition in C. *costFun_reward_value* and *costFun_penalty_value* are assigned according to the definition of cost function operators in the ML theory (according to the equation 3).

VII. EXPERIMENTAL RESULTS

Experiments are conducted on the BeagleBoard-xM (BBxM) embedded platform, which contains a TI OMAP DM3730 [24] SoC with an ARM Cortex-A8 processor. The platform runs Linux Operating System 3.7.10 together with the Ubuntu 12.04 distribution⁴. For the video decoder case study, we used FFMPEG⁵ libraries, running H.264 video⁶ of VGA resolution (640x480) for 720 frames, which at 23.976 FPS is 30.03 seconds. In order to send the Application Annotations, the H.264 decoder code was modified to include the API functions: *config_governor(int fps)*, *start_governor()*, *new_epoch()* and *stop_governor()*, which use *ioctl* to trigger the Governor (Section VI-A).

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 I. DM3730 SPECIFICATIONS (ARM CORTEX-A8) [25]

A. Performance and Power Consumption

Figure 6 shows the runtime performance of the code generated RTM, where the first plot shows the performance throughout each frame. Maximum performance is capped at 23.976 FPS because the platform starts decoding the next frame keeping up with the given FPS. This means that a frame decoded under $1/23.976\text{fps} = 41.7\text{ms}$ will produce a positive reward. The second plot shows the V-F decisions and the power consumption in turn for each frame. This shows the exploration phase of the RTM at the beginning of the video. It can be seen during the exploration phase, low V-F settings tend to cause performance losses. A second exploration phase is carried out at roughly the 330th frame, so the algorithm can take better decisions for the second half of the video, with less performance penalties. This behaviour is comparable to the one

presented in [15], where exploration and exploitation phases are present, with more variations at early stages of the runtime. Power consumption has been estimated using Table I, which lists the CPU power specifications for this architecture. Four frequencies are available: 300MHz, 600MHz, 800MHz, 1GHz. Energy consumption for this code generated architecture is estimated to be $16\mu\text{J}$. Energy consumption for the code generated RTM is lower than that of the *Ondemand* governor by 4%. This is the first step for the RTM code generation, which is continuing work to optimise its performance.

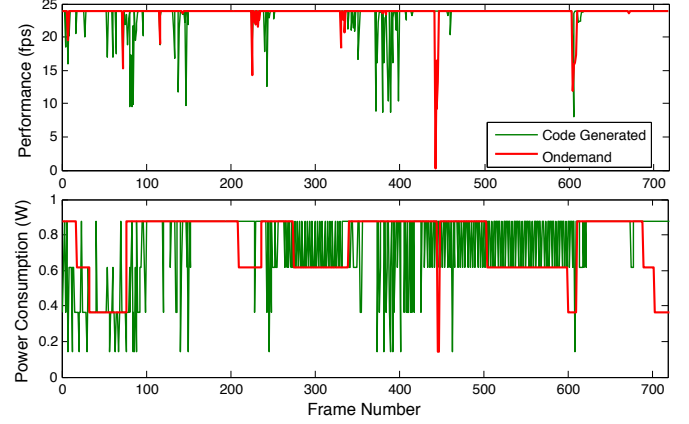


Fig. 6. Performance and Power Consumption of the generated RTM Governor for an H.264 Video

B. Overheads

The RTM Linux Governor implementation uses a total of 13.5kB of RAM when loaded, this including the auto generated code and the LKM interfaces (Freq. Changer, Perf. Monitor). The algorithm takes on average 39K clock cycles to run, which, at lowest frequency (300MHz) would be 0.13ms of timing overhead. Including the frequency change overheads, the algorithm takes 188K clock cycles, which would take 0.63ms. Comparing this overhead with the video decoder, its FPS are of 23.976 which translates to 41.7ms, so the algorithm overhead (including frequency changes) is 1.51%.

C. Discussion

Comparing this work with our first experiment of developing a hand-coded RTM system for a video decoder application in [15], we found out how formal modelling and automatic code generation can reduce the effort needed at the implementation level. The development of the hand-coded RTM [15] at kernel level was not trivial for obtaining the knowledge of kernel drivers, interfaces and user space communication for the platform used, plus the time needed for the development of the algorithm. In this work we tackle this problem by separating the RTM governor interface and the RTM algorithms, since the RTM governor interface is developed once and can be used for different RTM algorithms. Moreover, debugging at kernel level is a challenging task. We overcome these issues by formal verification at the Event-B abstraction level before implementation, ensuring that the RTM algorithms behaviour is correct independent of the platform. For further deployment in a different platform, the governor interface will be modified for that specific platform, whilst the code generated RTM algorithms are unchanged.

⁴Nelson: <https://eewiki.net/display/linuxonarm/BeagleBoard>

⁵FFMPEG: <http://www.ffmpeg.org>

⁶Big Buck Bunny: <https://peach.blender.org/>

VIII. CONCLUSIONS AND FUTURE WORK

We have proposed a formal approach toward automatic code generation of RTM systems from a verified Event-B model. This work has several contributions, first, this is a novel model-based method for developing RTM algorithms automatically. Second, the models are formal and thus amenable to formal verification, addressing reliability and correctness of the models.

We have modelled a RTM system for a video decoder application and we experiment executing our automatically generated codes as a Linux governor. The experimental results indicate that our model-based approach produces code with an acceptable performance overhead. In implementation, we have proposed a generic approach by separating the RTM interface (includes interfaces to interact with the application and hardware) and code generated RTM (includes the algorithms). Our approach will make it easier to re-target at other architectures and applications, since it is easier to manage and verify model evolution than code evolution. Moreover, decomposing our model into the controller and environment sub-models will increase reusability by separating RTM algorithms (controller) from hardware and application interfaces (environment). As a future work, the objective is to deploy the code generated RTM system into different architectures. This will be done by modifying the underlying governor interface for each architecture, whilst the code generated RTM algorithms remain unchanged.

In order to gain a better energy saving, optimisation of the algorithm parameters is required. This optimisation includes the comparison of different prediction algorithms and selecting the most efficient one. A distinct difference between the algorithm presented here and the one in [15], is that the latter does prediction by separating workload types, which reduces prediction error. This feature will be addressed in further refinement of the Event-B model. Finally, the last refinement model can be compared with the State-of-the-Art available.

ACKNOWLEDGMENT

This work was supported in part by an Engineering and Physical Sciences Research Council Programme Grant, EP/K034448/1 (See www.prime-project.org for more information) and by Consejo Nacional de Ciencia y Tecnología (CONACYT) of Mexico, grant 213977.

REFERENCES

- [1] K. Choi, W.-C. Cheng, and M. Pedram, "Frame-Based Dynamic Voltage and Frequency Scaling for an MPEG Player," *JOLPE*, vol. 1, pp. 27–43, Apr. 2005.
- [2] S.-y. Bang, K. Bang, S. Yoon, and E.-y. Chung, "Run-Time Adaptive Workload Estimation for Dynamic Voltage Scaling," *IEEE TCAD*, vol. 28, Sept. 2009.
- [3] Y. Gu and S. Chakraborty, "Control theory-based DVS for interactive 3D games," in *DAC*, (New York, New York, USA), p. 740, ACM Press, 2008.
- [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*, vol. 47, pp. 143–162, Jan. 2011.
- [5] M. Moeng and R. Melhem, "Applying statistical machine learning to multicore voltage & frequency scaling," in *Proceedings of the 7th ACM international conference on Computing frontiers*, (New York, New York, USA), pp. 277–286, ACM, 2010.
- [6] J. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
- [7] J. Abrial, "Formal Methods: Theory Becoming Practice," *J. UCS*, vol. 13, no. 5, pp. 619–628, 2007.
- [8] M. Keating, D. Flynn, R. Aitken, and K. Shi, *Low power methodology manual: for system-on-chip design*. Springer Verlag, 2007.
- [9] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, 2006.
- [10] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *ISLPED*, (New York, New York, USA), pp. 207–212, ACM Press, 2007.
- [11] G. Dhiman and T. Rosing, "System-level power management using online learning," *IEEE TCAD*, vol. 28, pp. 676–689, May 2009.
- [12] A. Sinha and A. Chandrakasan, "Dynamic voltage scheduling using adaptive filtering of workload traces," in *VLSI Design*, pp. 221–226, IEEE Comput. Soc, 2001.
- [13] S. Sinha, J. Suh, B. Bakkaloglu, and Y. Cao, "Workload-Aware Neuromorphic Design of the Power Controller," *IEEE JETCAS*, vol. 1, pp. 381–390, Sept. 2011.
- [14] P. Bose, a. Buyuktosunoglu, J. a. Darringer, M. S. Gupta, M. B. Healy, H. Jacobson, I. Nair, J. a. Rivers, J. Shin, a. Vega, and a. J. Weger, "Power management of multi-core chips: Challenges and pitfalls," in *DATE*, pp. 977–982, 2012.
- [15] L. A. Maeda-Nunez, A. K. Das, R. A. Shafik, G. V. Merrett, and B. Al-Hashimi, "Pogo: an application-specific adaptive energy minimisation approach for embedded systems," in *HIPEAC Workshop on Energy Efficiency with Heterogeneous Computing*, 2015.
- [16] A. S. Fathabadi, C. F. Snook, and M. J. Butler, "Applying an Integrated Modelling Process to Run-time Management of Many-Core Systems," in *IFM 2014, Bertinoro, Italy, September 9-11, Proceedings*, pp. 120–135, 2014.
- [17] A. Edmunds and M. J. Butler, "Linking event-b and concurrent object-oriented programs," *Electr. Notes Theor. Comput. Sci.*, vol. 214, pp. 159–182, 2008.
- [18] G. E. P. Box, "Understanding Exponential Smoothing – A Simple Way to Forecast Sales and Inventory," *Quality Engineering*, 1990.
- [19] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, vol. 28. Cambridge Univ Press, 1998.
- [20] J. Abrial, M. J. Butler, S. Hallerstede, and L. Voisin, "An Open Extensible Tool Environment for Event-B," in *Formal Methods and Software Engineering, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, pp. 588–605, 2006.
- [21] A. Salehi Fathabadi, M. Butler, and A. Rezazadeh, "Language and tool support for event refinement structures in Event-B," *Formal Aspects of Computing*, pp. 1–25, 2014.
- [22] S. Hallerstede, "On the Purpose of Event-B Proof Obligations," in *ABZ, London, UK*, pp. 125–138, 2008.
- [23] ARM, "ARM Cortex-A8 Reference Manual," 2010.
- [24] Texas Instruments, "DM3730, DM3725 Digital Media Processors Datasheet," 2011.
- [25] Texas Instruments, "AM/DM37x Power Estimation Spreadsheet," 2011.

Learning Transfer-based Adaptive Energy Minimization in Embedded Systems

Rishad A. Shafik, Anup Das, Luis A. Maeda-Nunez, Sheng Yang, Geoff V. Merrett & Bashir M. Al-Hashimi

Abstract—Embedded systems execute applications with different performance requirements. These applications exercise the hardware differently depending on the types of computation being carried out, generating varying workloads with time. We will demonstrate that energy minimization with such workload and performance variations within (intra) and across (inter) applications is particularly challenging. To address this challenge we propose an online energy minimization approach, capable of minimizing energy through adaptation to these variations. At the core of the approach is an initial learning through reinforcement learning algorithm that suitably selects the appropriate voltage/frequency scalings (VFS) based on workload predictions to meet the applications’ performance requirements. The adaptation is then facilitated and expedited through learning transfer, which uses the interaction between the system application, runtime and hardware layers to adjust the power control levers. The proposed approach is implemented as a power governor in Linux and validated on an ARM Cortex-A8 running different benchmark applications. We show that with intra- and inter-application variations, our proposed approach can effectively minimize energy consumption by up to 33% compared to existing approaches. Scaling the approach further to multi-core systems, we also show that it can minimize energy by up to 18% with 2X reduction in the learning time when compared with a recently reported approach.

I. INTRODUCTION

Energy minimization is a prime design objective for embedded systems. To enable energy minimization these systems are equipped with processors with dynamic voltage and frequency scaling (DVFS) capabilities, controlled by the operating system (OS) and system firmware; examples include Linux’s power governors and ARM’s VFS firmware [1]. The basic principle is to reduce the operating voltage/frequency (V/F) dynamically at runtime, resulting in a cubic decrease in power consumption at the expense of a linear performance degradation [2], [3], [4].

Dynamic energy minimization approaches can be broadly classified into two types – offline and online. The offline approach characterizes the workloads of a given application exploiting application-specific knowledge. The profiled workloads are then used during runtime to adjust the power control levers at regular intervals for achieving energy minimization. Examples include workload characterization and energy minimization of video decoders [5], [6] and control-theoretic formulation of energy consumption of multimedia workloads [7].

Energy minimization using online approach has the basic principle of controlling the hardware power levers based on the processor workloads [8], [9]. Depending on the control mechanism, the online approach can be reactive or proactive.

In the reactive approach the VFS is controlled based on the history of hardware CPU usage. When the CPU usage is higher/lower than a pre-defined value, an increased/decreased V/F is used. Linux’s ondemand power governor [11] is a typical example of such an approach. In the proactive approach predicted workloads are used to manage the hardware power control levers [10], [12]. The impact of such control is then observed through hardware performance monitors to evaluate the actual processor workload and adjust the future controls to account for any mispredictions. Jung *et al.* [13] proposed one such approach using an initial value problem based processor workload prediction and classification. The predicted workload is then used for continuous V/F adjustment to achieve energy minimization. To observe the impact of such V/F adjustment on performance, information from the processor’s performance monitoring unit are used. Ramakrishna *et al* [14] showed an online prediction-based approach to classify the task workloads and apply VFS suitably using the feedback from the CPU performance monitors.

Since processor workloads are exercised differently depending on application tasks, Siyu *et al.* [15] and Shen *et al.* [17] have proposed online prediction-based control approaches of hardware power levers using machine learning algorithm. Their approaches have shown methods of workload classification for suitably learning the VFS required for an application to achieve energy minimization in the presence of performance variations due to application-generated CPU workloads. However, these approaches do not consider the variation of application performances, such as frame rate for video decoders, page loading rate for browsers etc. Among others, learning-based idle-time manipulation has been proposed in [18] to reduce energy in multi-core systems.

Modern embedded systems feature workload and performance variations both within and across applications [16]. Such variations can arise due to the types of computation being carried. Energy minimization with such variations is challenging using the existing online approaches (see Section II). This is because the online approaches, such as [11], [14], [17] do not interact with the applications for their changing performance needs, which leads to either over-performance or failure in meeting their performance requirements. Moreover, existing approaches using machine learning [15], [17], [18] use a single runtime formulation of V/F scaling for a given performance requirement, which cannot adapt to intra- and inter-application workload and performance variations. To effectively minimize energy consumption in the presence of such workload and performance variations, this paper makes the following specific **contributions**:

- an adaptive energy minimization approach to effectively detect and deal with the intra- and inter-application workload and performance variations,
- a reinforcement learning (RL) algorithm to suitably select

the appropriate VFS for a given performance requirement, followed by a transfer learning algorithm to adapt to such workload and performance variations, and

- a Linux power governor implementation of the approach for extensive validation using different benchmark applications.

To the best of authors' knowledge, this is the first complete work that shows transfer learning-based adaptive energy minimization and its implementations on single and multi-core embedded systems. The remainder of this paper is organized as follows. Section II motivates the proposed approach, Sections III describes the approach and its implementation. Sections IV and V reports the experimental results and analyzes the learning overheads, Section VI demonstrates scaling of the approach to multi-core systems. Finally, Section VII concludes the paper.

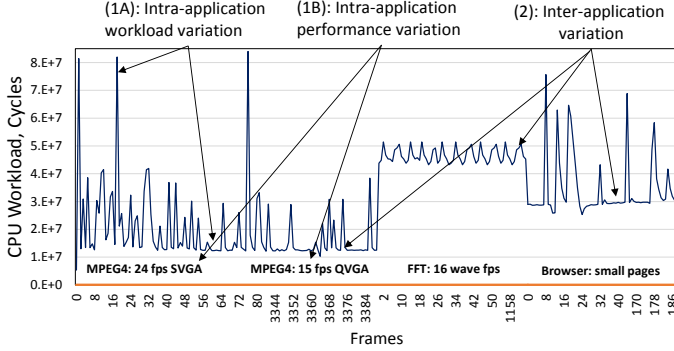


Fig. 1. Workload and performance variations within and across applications

II. MOTIVATION

Fig. 1 shows an example of workload and performance variations within (intra) and across (inter) applications. It highlights the application performances and observed CPU workloads (in CPU cycles) of three different applications: MPEG4 [20] followed by FFT [21] and browser (based on [22]) rendering small html pages (less than 20kb in size) from *bbench* [23]. The CPU workloads were recorded on a DM3730 system-on-chip from Texas Instruments, integrated on the BeagleBoard-xM (BBxM) platform [19], which incorporates an ARM Cortex-A8 CPU core running at 800MHz CPU clock speed. From Fig. 1 the following two observations are made:

Observation 1: The CPU workload and performance requirements vary within an application. For example, MPEG4 decoding at 24 SVGA frames per second (fps) exhibits up to 7x workload variation, Fig. 1 (1A). Such variation arises due to decoding of intra-coded (I) SVGA frames with higher computations, followed by a number of predictive-coded (P) frames with lower computations [27]. The MPEG4 also experiences a performance change from 24 SVGA fps to 15 QVGA fps for the given workload profile, Fig. 1 (1B).

Observation 2: The CPU workload changes when the system switches between applications. Referring to Fig. 1(2), when the system switches from MPEG4 to FFT, the workload increases by 3x on average, since FFT wave frames are computationally intensive. Also, when the system switches from FFT to browser, the workload decreases by 2x due to less computations required per small page rendering. Due to such switches the performance requirements also change from 67 ms per MPEG4 frame to 62 ms per FFT wave frame and then to 100 ms per browser page.

Minimizing energy consumption while meeting application performance requirements can be particularly challenging with

the above-mentioned variations. Because the processor power control levers will need to be continually adjusted to suitably select the V/F scaling in the presence of these variations. To achieve such adjustment effectively through online energy minimization the following key aspects need to be addressed:

- *Learning:* The relationship between hardware power control levers and applications' performance requirements need to be learnt with an aim to suitably select the appropriate VFS, while meeting the application performance requirements.
- *Adaptation:* When the CPU workload or the application performance requirement changes the power control levers must be adapted accordingly. To enable such adaptation, the changes in performance requirements must be communicated from the application to the runtime (i.e. the system software including OS scheduler and VFS controller), and the workload variations must be monitored through the hardware performance counters.

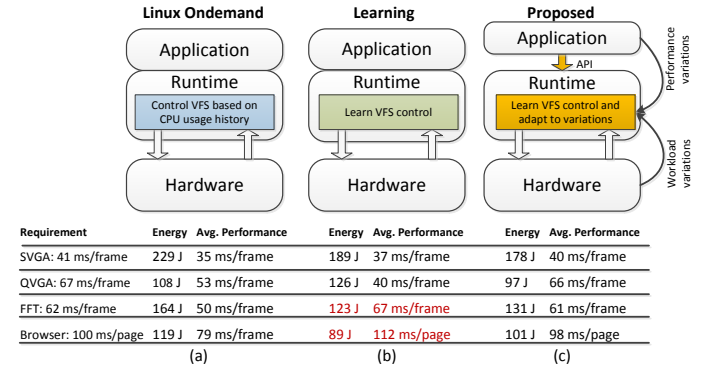


Fig. 2. Comparison between energy minimization approaches: (a) Linux's ondemand [11], (b) learning-based [17], and (c) proposed

This paper proposes a learning transfer-based adaptive energy minimization approach, addressing the above-mentioned aspects. To highlight the importance of such adaptation, Fig. 2 shows three energy minimization approaches applied to the application scenarios in Fig. 1: MPEG4 decoding at 24 SVGA fps (41 ms/frame) and 15 QVGA fps (i.e. 67 ms/frame), FFT processing 16 wave fps (i.e. 62 ms/frame) and browser rendering at 100 ms/page. The energy consumption incurred by the applications are measured using an Agilent DC Power Analyzer (N6705B) (see Section IV for details).

The learning approach (Fig. 2.(b)) carries out a single formulation of the VFS controls based on the predicted workloads using machine learning [17]. However, as the VFS controls resulting from such learning approach do not adapt to workload and performance variations across the applications, it cannot achieve effective energy minimization. For example, after initially learning VFS controls for the MPEG4 decoding at 24 fps, the learning approach over-performs and incurs higher energy consumption for the MPEG4 decoding 15 QVGA fps. Conversely, in the case of FFT and browser applications it under-performs and violates the specified performance requirements (highlighted in red). Due to performance-agnostic nature of VFS controls the Linux ondemand over-performs for most of the applications, incurring higher energy consumptions (Fig. 2.(a)). The proposed approach provides the lowest energy consumption for all intra- and inter-application variations. This is because it learns the appropriate VFS controls and adapts to performance and workload variations across all applications (Fig. 2.(c)).

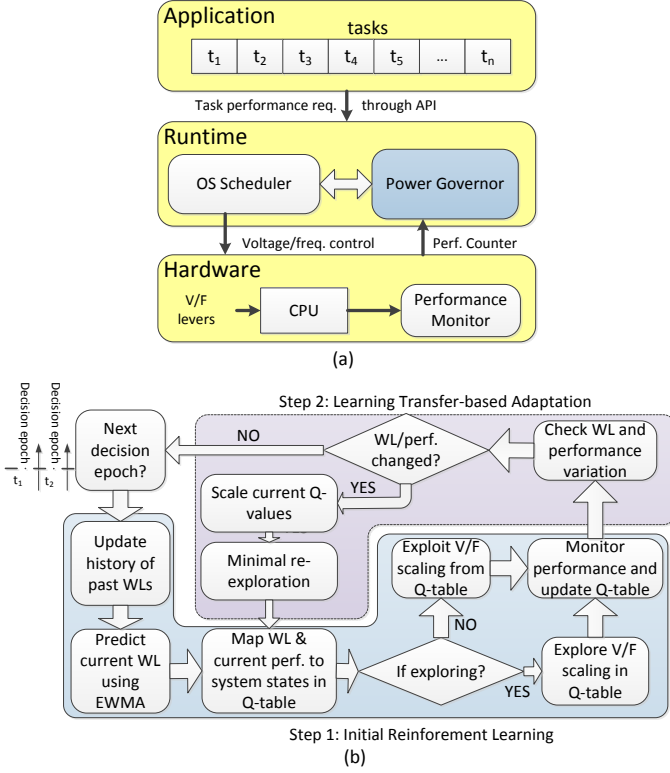


Fig. 3. (a) Proposed adaptive approach showing interactions between layers, (b) flowchart of the proposed adaptive energy minimization approach

III. PROPOSED ENERGY MINIMIZATION APPROACH

The proposed approach is shown in Fig. 3.(a) highlighting the interactions between application, runtime and hardware layers. The application layer consists of a series of computation tasks being executed at time intervals (we refer to these as decision epochs), each with a specified performance requirement. The performance requirement is communicated to the runtime layer through an application programming interface (API) as below

```
rts.set_perf(41);
```

where *rts* is a thread-safe runtime variable within the API (see Appendix A), *set_perf* sets the performance requirement as 41 ms per decision epoch. The runtime layer consists of the power governor implementing the proposed adaptive energy minimization and the OS scheduler (not implemented, shown for completeness). With the given performance requirement, the power governor controls and adapts the appropriate hardware power levers (i.e. VFS) at regular decision epochs, while meeting the specified application performance requirement.

Fig. 3.(b) shows a flowchart of the proposed energy minimization approach, organized in two major steps: reinforcement learning and learning transfer. The reinforcement learning sets up proactive VFS controls at each decision epoch through state prediction. Based on the predicted states, the VFS controls are learnt to meet the specified performance requirement. When performance or workload variations are detected, these controls are adapted through a learning transfer algorithm with an aim of ensuring energy minimization in the presence of such variations. These steps are detailed in the following.

A. Step 1: Reinforcement Learning

The initial learning through reinforcement learning algorithm evolves in three phases, as follows.

1) *State Prediction and Q-table Formation*: State prediction is a required phase in the reinforcement learning (RL) step to identify the Q-value of the future system state. In our proposed approach, we also use the same predictor to classify the expected workload to a system state at the beginning of each decision epoch, as also used in [14], [15], [17]. This predictor estimates the current CPU workload based on the history of the past workloads and maps this CPU workload to a system state based on the current performance. The CPU cycles' count is preferred as the workload parameter over the other parameters, such as memory accesses, cache misses and instruction rate, etc. as it directly defines the CPU activity due to processor executing instructions of a given computing task. To predict the workload, an exponential weighted moving average (EWMA) scheme is used, similar to [5], [8]. Using this scheme, the predicted workload for the $t + 1^{th}$ decision epoch, \hat{C}_{t+1} is

$$\hat{C}_{t+1} = \omega C_t + \sum_{i=t-D}^{t-1} (1-\omega)^i C_i, \quad (1)$$

where C_t and C_i are the previous observed workloads (in CPU cycles) at the t^{th} and i^{th} decision epochs, $(t-D) \leq i \leq (t-1)$, ω is the moving average coefficient and D is the window size of past observed workloads. The ω and D values are chosen to give higher prediction accuracy for the given application workloads, similar to [14]. However, the workload prediction through (1) still undergoes mispredictions during runtime due to variations in workloads. The impact of such mispredictions on the state prediction and corresponding VFS controls are discussed in Section IV-A.

The predicted system state is determined by using the estimated workload through (1) and the current performance as a pair. Hence, the system state space \mathcal{S} is comprised of the combinations of current performance state in terms of the average slack ratios (\mathcal{L}) and the predicted CPU workloads (\hat{C}), denoted as $\mathcal{S}\{\mathcal{C}, \mathcal{L}\}$. For each state (s_t), the average slack ratio at t_{th} decision epoch (\mathcal{L}_t) is divided in five bins (as an example) as

$$\forall_{s_t} : \begin{cases} \mathcal{L}_t > 0.15 \\ 0.05 < \mathcal{L}_t \leq 0.15 \\ |\mathcal{L}_t| \leq 0.05 \\ -0.15 < \mathcal{L}_t \leq -0.05, \& \\ \mathcal{L}_t \leq -0.15, \end{cases} \quad (2)$$

Similarly, for each state the CPU workloads are divided in several workload bins, each with ranges (i.e. ΔC) around a base workload (C_b). As an example illustration, workload states with six bins is as follows:

$$\forall_{s_t} : \begin{cases} \hat{C}_{t+1} > (C_b + 2\Delta C), \\ (C_b + 2\Delta C) > \hat{C}_{t+1} \geq (C_b + \Delta C), \\ (C_b + \Delta C) > \hat{C}_{t+1} \geq C_b, \\ C_b > \hat{C}_{t+1} \geq (C_b - \Delta C), \\ (C_b - \Delta C) > \hat{C}_{t+1} \geq (C_b - 2\Delta C), \\ \hat{C}_{t+1} < (C_b - 2\Delta C). \end{cases} \quad (3)$$

With the given combinations between \mathcal{L}_t and \hat{C}_{t+1} in (2) and (3), state entries for the Q-table are set up. Thus, for each predicted workload (\hat{C}_{t+1}) and the current performance (\mathcal{L}_t) pair the system state is mapped using (2) and (3).

The state space, and the action space (formed of the VFS control options, denoted as $\mathcal{A}\{Vdd, f\}$) define the size of Q-table for the RL step. The size of the Q-table in terms of the total number of state-action pairs ($|\mathcal{A}\{Vdd, f\}| \times |\mathcal{S}\{\mathcal{C}, \mathcal{L}\}|$) is

important for the RL algorithm as it influences the trade-off between learning overhead and energy minimization achieved (see Section III-B and IV for detailed trade-offs). In this work, the size of Q-table is carefully chosen to ensure a good trade-off between learning overhead and energy minimization (see Section V). With the given state prediction and Q-table formation, the RL algorithm carries out exploration and evaluation of the VFS control actions as discussed next.

2) *Exploration*: Traditionally, exploration in a Q-table is carried out using a random action selection strategy from the pool of actions, each with a uniform probability distribution. However, such exploration is inefficient as it does not use the intuitive relationships that often exist between the state-action pairs [24]. For example, when the system is currently in a state with higher positive average slack ratio (\mathcal{L}), lower operating frequency actions are likely to reduce it. Similarly, when the system is in a state with higher negative \mathcal{L} , higher operating frequency actions are needed to converge close to zero \mathcal{L} values. To reflect such relationship during exploration, we use the following discrete exponential probability distribution function for the selection of action (a_t)

$$p(a_t)_{a_t \in \mathcal{A}\{Vdd, f\}} = \lambda \exp[-\lambda f(a)\beta\mathcal{L}], \quad (4)$$

where λ is the uniform probability of actions (i.e. $\lambda = 1/|\mathcal{A}\{Vdd, f\}|$), $f(a)$ is the operating frequency in action a and β is a constant. According to (4) when \mathcal{L} is close to zero, the exploration probabilities are almost uniform, guided by λ . However, positive and negative \mathcal{L} prioritize selection of lower and higher frequencies, respectively. The discrete exponential probability distribution, given by (4), has an advantage in terms of quicker learning, which can be stated and proven by Lemma I (see Appendix B). Upon selection of the action, the resulting performance impact is evaluated in terms of the Q-value. At the beginning of the $(t+1)^{th}$ decision epoch, the Q-value corresponding to a selected VFS action is updated as [15]:

$Q(s_{t+1}, a_t) = Q(s_t, a_t)(1-\alpha) + \alpha[r_t + \gamma \max_{a_t} Q(s_{t+1}, a_t)]$, (5) where α is the learning rate, γ is the discount factor to de-scale the current maximum Q-value in the row ($0 \leq \alpha, \gamma \leq 1$), s_t is the observed state in the t^{th} decision epoch and s_{t+1} is the predicted state in the $(t+1)^{th}$ decision epoch determined by estimated workload and current performance (see Section III-A1). The reward function r_t in (5) is computed as a function of the resulting performance in terms of average slack ratio at the t^{th} decision epoch (\mathcal{L}_t) and change in the average slack ratio since last decision epoch ($\Delta\mathcal{L}$), given by

$$r_t = K_1|\mathcal{L}_t| + K_2\Delta\mathcal{L}, \quad (6)$$

where K_1 and K_2 denote constant values for different bins of average slack ratios in (2). These values are pre-determined such that higher rewards are offered to actions that reduce the \mathcal{L}_t to near zero (i.e. within $\pm 5\%$, chosen as range for soft deadlines for applications) values or to actions that show improved $\Delta\mathcal{L}$ (i.e. from a higher to lower slack ratio). The K_1 values also ensure that negative rewards are awarded to actions that result in increased average slack ratio ($\Delta\mathcal{L}$). The $\Delta\mathcal{L}$ values in (6) are estimated at each decision epoch by

$$\mathcal{L}_t = \frac{1}{N(T_{ref})} \sum_{i=0}^t (T_{ref} - T_i - T_{OVH}), \quad (7)$$

where T_{ref} is the real-time reference for execution time per decision epoch, T_i is the application task execution time incurred due to choice of VFS actions at every i^{th} decision epoch ($0 \leq i \leq t$), N is the number of decision epochs elapsed

since application started for a given T_{ref} , T_{OVH} is the sum of overheads caused by VFS and reinforcement learning (evaluated empirically through offline profiling, see Section V) and $\Delta\mathcal{L}_t$ is average slack ratio difference since last observation, given as $\Delta\mathcal{L}_t = \mathcal{L}_{t-1} - \mathcal{L}_t$. The T_i in (7) can be estimated at every decision epoch as the ratio between the observed processor CPU cycles (C_i) from its performance monitors and operating frequency chosen (f_i) at i^{th} decision epoch as

$$T_i = \frac{C_i}{f_i}. \quad (8)$$

Equations (5) and (6) set up the reinforcement learning-based exploration of VFS control actions.

3) *Exploitation*: After the VFS control actions for the Q-table states have been explored and evaluated, the RL algorithm enters into an exploitation phase. The transition from the exploration to exploitation phase is controlled through the exploration probability (EP), denoted by ϵ ($0 \leq \epsilon \leq 1$). To accelerate exploitation ϵ_t at the t^{th} decision epoch is updated as

$$\epsilon_t = \epsilon_{t-1} \exp[-(1-\alpha)], \quad (9)$$

where α is the learning factor per decision epoch. Based on the ϵ_t value, the exploration or exploitation is carried out to find the best policy sub-set ($\pi^*(s_t, a_t)$) from a set of exploration policies ($\Pi(s_t, a_t)$, $\pi \in \Pi$) as follows:

$$\pi^*(s_t, a_t) = \begin{cases} a_t : \max(Q(s_t, a)); & \text{if } p > \epsilon_t, \\ a_k : p(a_k) \text{ is given by (4)} & \end{cases} \quad (10)$$

where p is a random value uniformly distributed over $[0, 1]$. As can be seen, when ϵ_t value decreases in (9), the probability of exploitation increases in (10).

TABLE I
THE COMPARISONS BETWEEN SINGLE Q-TABLE APPROACH [9] VERSUS
LEARNING TRANSFER APPROACH

$ \mathcal{A} $	$\Delta\mathcal{C}$	Q-table size: $ \mathcal{A} \times \mathcal{S} $ (single Q-table)	Q-table size: $ \mathcal{A} \times \mathcal{S} $ (learning transfer)
4	1×10^7	4000	120
4	2×10^7	2000	120
4	4×10^7	1000	120
6	1×10^7	6000	180
6	2×10^7	3000	180
6	4×10^7	1500	180

B. Step 2: Learning Transfer-based Adaptation

Using a single Q-table in RL algorithm step for covering the dynamic ranges of workload and performance variations can expand the learning space substantially. Table I shows example illustrations of the impact of using a single Q-table approach, similar to [9]; column 1 and 2 show the number of actions and the size of workload bins, while column 3 shows the number of state-actions pairs considering a workload coverage from 0 to 10^{10} cycles. As can be seen, to cover such a dynamic workload change a single Q-table will have 4000 state-action pairs with 4 actions with workload bin size of 10^7 each. The size of the Q-table can expand further to 6000 if the number of actions increases to 6.

To ensure a quicker learning and adaptation to dynamic workload or performance variations, smaller Q-table is used in this work together with learning transfer. Columns 5 and 6 show the motivation of using such learning transfer-based adaptation (Table I). As expected with a smaller state space of 30, the number of state-action pairs is 120 for 4 actions, and 180 for 6 actions around a base workload. With such smaller Q-tables the learning transfer also benefits from quicker convergence and learning of the Q-table. Table II compares the worst-case

convergence times between single Q-table approach and smaller Q-tables in the learning transfer-based approach considering both workload and performance variations in the case of $\Delta C=10^7$ and $|\mathcal{A}|=4$. Columns 1-3 show the number of workload variations and the corresponding number of decision epochs required for full convergence of both approaches. As can be seen, the single Q-table approach takes invariably 4000 decision epochs despite any workload variations for the full learning of the Q-table. The learning transfer-based approach takes significantly lower number of decision epochs for the same due to smaller Q-tables (Table I). The convergence time in this approach, however, depends on the number of workload variations. As can be seen, when the number of workload variation increases from 1 to 8 within a given time frame, the convergence time increases from 160 to 440 decision epochs, which is still significantly lower than the single Q-table approach.

Table II also compares the worst case convergence times of both approaches for different performance variations (columns 4-6). As can be seen, the single Q-table approach requires significantly high number of decision epochs when performance variations (i.e. change in the application task deadline, T_{ref}) are encountered. This is because the original Q-table can no longer provide the the optimized VFS scaling options with such variation in T_{ref} , which necessitates re-learning from scratch again. Unlike the single Q-table approach, the learning transfer can continue to exploit the learning from the previous T_{ref} and converge quickly to give the optimized VFS options for the given system states. The convergence times of the proposed learning transfer-based approach in the event of workload and performance variations can be validated through Lemma I (see Appendix B). Since learning transfer-based approach requires simpler re-definitions of state-space and minimal re-learning, the overhead is smaller compared to RL step (see Fig. 13).

TABLE II
COMPARATIVE CONVERGENCES OF SINGLE Q-TABLE APPROACH [9] AND
PROPOSED LEARNING TRANSFER-BASED APPROACH

Workload variation			Performance variation		
No. of Variations	Learning convergence		No. of Variations	Learning convergence	
	single	transfer		single	transfer
0	4000	120	0	4000	120
1	4000	160	1	8000	160
2	4000	200	2	12000	200
4	4000	280	4	20000	280
8	4000	440	8	36000	440

When the workload profile changes within an application beyond the current base workload (observation 1), the current Q-table fails to minimize energy consumption effectively. The same is also true when the performance requirement changes within and across applications (observation 2). To enable adaptation to these variations the proposed approach first detects these variations through inter-layer interactions (Fig. 4). This is then followed by a Q-table update through learning transfer algorithm (Algorithm 1). The detection of workload and performance variations and their adaptations are further detailed next.

1) *Adaptation to Workload Variation:* The workload variation is detected through the interaction between the runtime and the hardware layers in the following phases, as shown in Fig. 4. Initially, with a given performance requirement of T_{ref} per application task (Fig. 3.(a)), the runtime learns VFS control actions (phases 1-2). During this time the runtime also computes the short-term average workload, C_n , over the last n decision

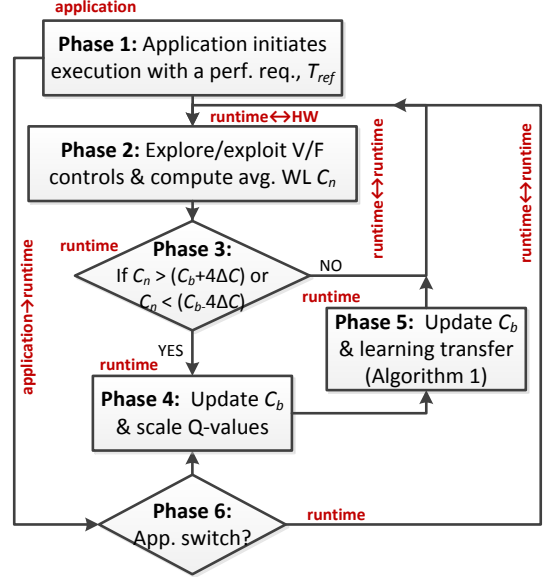


Fig. 4. Learning transfer-based adaptation to workload/performance variations epochs (the impact of varying n is studied in Section V). The mean workload, C_n , is then compared with the current base workload (C_b) in the Q-table (phase 3). If C_n is confirmed as beyond the current table limits (i.e. $C_n > (C_b + 4\Delta C)$ or $C_n < (C_b - 4\Delta C)$), the Q-table states are updated and scaled with a new base workload (C_b) nearest to C_n (phase 4). The scaling from the old Q-table is carried out as follows:

$$\forall_t : Q(s_t, a_t)_{scaled} = Q(s_t, a_t)_{old} \exp \left[-\frac{1}{\mathcal{L} \rho_{WL}} \right], \quad (11)$$

where ρ_{WL} is the scaling ratio proportional to $\frac{C_{b_{new}}}{C_{b_{old}}}$. Note that the Q-value scaling in (11) ensures that the Q-values are downscaled according to the \mathcal{L} states. At higher \mathcal{L} values the $Q(s_t, a_t)_{scaled}$ values are less downscaled, while at lower \mathcal{L} values the $Q(s_t, a_t)_{scaled}$ values are more downscaled to ensure that further exploration of VFS control actions can update the optimal policy π^* quickly. To minimize such exploration, the current best policy ($\pi^*(s_t, a_t)$) in the scaled Q-table is then updated in the next phase using learning transfer algorithm, as shown in Algorithm 1 (phase 5).

Algorithm 1 Learning transfer algorithm

Require: ρ_{WL} , $Q(s_t, a_t)_{scaled}$, $\pi^*(s_t, a_t)_{old}$

- 1: **for** each s_t in $Q(s_t, a_t)_{new}$ **do**
- 2: **if** s_t is explored **then**
- 3: **if** $\mathcal{L}(s_t)$ is near-zero (i.e. $\pm 5\%$) **then**
- 4: set: $f(\pi'(s_t, a_t)_{new}) = \rho_{WL} \times f(\pi^*(s_t, a_t)_{old})$
- 5: **else**
- 6: set: $f(\pi'(s_t, a_t)_{new}) = f(\pi^*(s_t, a_t)_{old})$
- 7: **end if**
- 8: map: $f(\pi'(s_t, a_t)_{new})$ to the nearest action a'_t in $Q(s_t, a_t)_{new}$
- 9: **for** every action in $Q(s_t, a_t)_{new}$ **do**
- 10: **if** action is a'_t **then**
- 11: swap $Q(s_t, a'_t)_{new}$ with $Q(s_t, a'_t)_{scaled}$
- 12: **else**
- 13: set: $Q(s_t, a_t)_{new} = \alpha Q(s_t, a_t)_{scaled}$
- 14: **end if**
- 15: **end for**
- 16: **else**
- 17: set: $Q(s_t, a_t)_{new} = Q(s_t, a_t)_{scaled}$
- 18: **end if**
- 19: **end for**
- 20: **return** $\pi'(s_t, a_t)_{new}$ and $Q(s_t, a_t)_{new}$

As can be seen, for each already explored state (s_t) the chosen frequency in the new policy ($f(\pi'(s_t, a_t)_{new})$) is obtained

through scaling by a factor of ρ_{WL} (lines 2-6). For a state with near-zero slack (i.e. $\pm 5\%$), such scaling requires multiplying the old chosen frequency by the scaling ratio (ρ_{WL}), while for a state with higher positive or negative slack the old chosen frequency is retained as the new chosen frequency. After such scaling of chosen frequencies, their corresponding actions are mapped in the new Q-table and the new Q-values are updated through transfer of the scaled Q-values (lines 9-15). The Q-value of the new chosen action is set as the Q-value of the old chosen action (line 11), while the other values retained from the scaled Q-values through (11). For un-explored or partially explored state (s_t), the scaled Q-value is retained in the new Q-table (line 17). The resulting Q-table (line 20) with transferred learning is then used with a reduced exploration probability (ϵ_t) for accelerated re-exploration, instead of learning from the scratch.

2) *Adaptation to Performance Variation*: When the application performance requirement changes due to intra- or inter-application switch, the adaptation is enabled through the API-based interaction between application and runtime layer (phase 6, Fig. 4). Upon such interaction, the runtime layer learns the new T_{ref} with the old C_b and carries out Q-value scaling and learning transfer (phases 4 and 5). Similar to (11), the Q-value scaling is carried out as

$$\forall_t : Q(s_t, a_t)_{scaled} = Q(s_t, a_t)_{old} \exp \left[-\frac{1}{\mathcal{L} \rho_T} \right], \quad (12)$$

where ρ_T is the scaling ratio, proportional to $\frac{T_{ref,old}}{T_{ref,new}}$. Similar to (11), Equation (12) also scales $Q(s_t, a_t)_{old}$ values based on the \mathcal{L} values. Following the Q-value scaling in (12), the Q-values are transferred between action pairs using the Algorithm 1 with p_{WL} values replaced by p_T . The transferred Q-values are then used for further minimal explorations (Fig. 3.(b)). The learning transfer-based adaptation has the advantage of lower number of re-explorations required when compared with the re-learning based approach. The reduction of the number of explorations is described through Proposition I (see Appendix B).

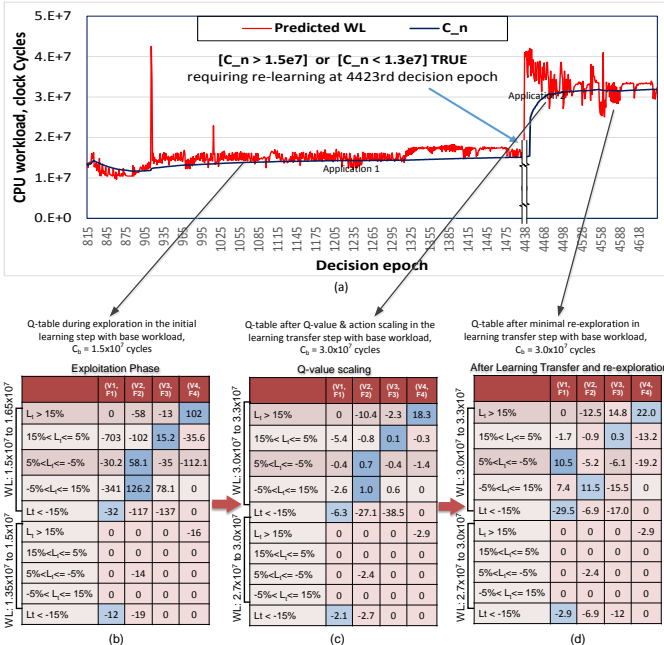


Fig. 5. Example illustration of learning transfer-based adaptation

3) *Example Illustration*: To illustrate how the proposed approach adapts to inter-application switch (from Application 1

to Application 2) as an example, Fig. 5.(a) plots the predicted (\hat{C}_t) and observed mean workload (C_n) values, with $n=300$ decision epochs, while Fig. 5.(b)-(d) show Q-tables for ten states covering two workload bins around the mean workload and five slack state variations as shown in (eq:statedef1) out of total 30 states (for demonstration purposes) and four actions for both applications. As can be seen in Fig. 5.(b), after the initial learning and exploration the Q-table is populated with two different kinds of states: explored states and partially explored or un-explored states. The explored states are more frequently visited due to workloads exercised by the hardware due to Application 1. As a result, most of the actions are evaluated and the best actions are chosen as the highest Q-value in the table (highlighted in blue). The un-explored or partially explored states are not visited as often and hence not all actions are evaluated (un-explored actions are marked by zero values). The proposed approach continues to exploit the learnt DVFS control decisions from the Q-table for the Application 1, which has a Q-table base workload of $C_b=1.5 \times 10^7$.

After the 4423rd decision epochs application switch happens from Application 1 to Application 2. Such inter-application variation causes both workload and performance variations. The workload variation is observed through the mean workload C_n , while performance variation is directly communicated by the application to the runtime (Fig. 4). To adapt to such variations, the proposed approach carries out learning transfer in two stages. First, the Q-values are scaled through (11) using the new base $C_b=3.0 \times 10^7$ near the current mean workload (C_n). The resulting scaled Q-values are shown in Fig. 5.(c). The scaled Q-values are then used to carry out further action transfer of explored and un-explored states through Algorithm 1. For explored states, the new actions new zero values ($\pm 5\%$) are further minimally re-explored. The resulting Q-values and the chosen actions after such exploration are shown in Fig. 5.(d). Due to such minimal re-exploration, the adaptation to workload and performance variations is much faster (see Tables I and II, and Section V). It is to be noted that after learning transfer and re-exploration only 1 out of 10 states is updated compared to re-learning based approach which would require updating all 10 states.

IV. EXPERIMENTAL RESULTS

The proposed adaptive energy minimization approach is implemented as a power governor in Linux kernel revision 3.7.10 (see Appendix A) running on DM3730 system-on-chip, integrated on the BeagleBoard-xM (BBxM) platform [19]. The platform consists of, among others, a single-core ARM Cortex-A8 CPU core, which supports four V/F levels: 300MHz at 0.93V, 600MHz at 1.10V, 800MHz at 1.26V and 1GHz at 1.35V [19]. To evaluate the effectiveness of the proposed governor, ffmpeg-based multimedia [20], MiBench benchmark [21] and browser [22] applications are executed. The energy consumptions of the ARM Cortex-A8 core are measured through direct observation of current and voltage using an Agilent DC Power Analyzer (N6705B). The current was observed by lifting an inductor off of the board and re-routing the signal through the same inductor and the power analyzer, while the voltage supplied across the Cortex-A8 was measured directly across the core supply. All experiments are carried out using a Q-table size of (30x4) consisting of 5 slack states and 6 workload

states as such table size gives the best trade-off between energy minimization and learning overheads as discussed in Section V.

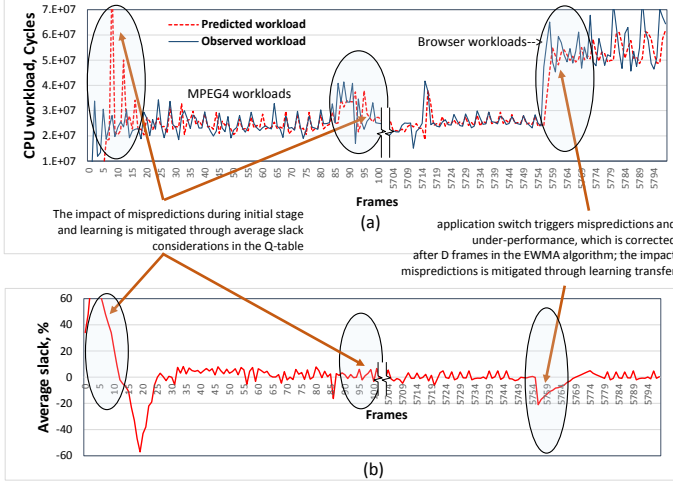


Fig. 6. (a) CPU workload predictions of MPEG4, followed by FFT, (b) impact of learning on the average slack ratio (in %)

A. Impact of State Prediction and Exploration

To investigate into the impact of state prediction on learning, Fig. 6.(a) shows the predicted and observed workloads (in CPU cycles), while Fig. 6.(b) shows the corresponding average slack ratios (\mathcal{L}) caused by the RL algorithm (Section III-A) for MPEG4 decoding at 24 SVGA fps, followed by browser application. As can be seen, the EWMA based workload prediction (given by (1)) incurs occasional mispredictions: during the exploration (the first 25 frames) and exploitation phases (after 90 frames) of MPEG4 and also briefly when the system switches from MPEG4 to browser (after 5750 frames, Fig. 6.(a)). The highest average misprediction of about 8% on average with respect to the mean observed workload was observed during the initial 100 frames in the MPEG4 decoding 24 SVGA fps, while the lowest misprediction of only 3% was observed for the following frames. To mitigate the impact of such mispredictions the RL algorithm (Section III-A) considers the current performance offset in terms of average slack ratio (\mathcal{L}_t) together with the currently predicted workload during state mapping of the next state. In the event of performance offset (showing positive or negative high \mathcal{L}_t) caused by mispredictions, the RL algorithm learns and applies the appropriate VFS controls to minimize it through the action rewarding mechanism (see Section III-A2). Similar to MPEG4, workload misprediction is also observed when the system switches to the FFT application after the 5754th frame. At this time the learning transfer and further explorations take place, which causes under-performance initially for about 45 FFT frames. However, after further explorations during the next 50 frames, the under-performance is offset by updating the learning of the appropriate VFS controls.

To highlight the advantages of the explorations using exponential probability distribution during the initial learning step (i.e. RL step), Table III compares the average number of explorations required by the proposed approach and the that of the learning-based approach [17]. Column 1 shows the applications with the input sizes used, while Columns 2 and 3 show the number of explorations recorded for the proposed and learning-based approaches. As can be seen, the proposed

TABLE III
COMPARATIVE NUMBER OF EXPLORATIONS DURING THE INITIAL LEARNING IN THE PROPOSED AND LEARNING APPROACHES

Application	No. explorations (proposed)	No. explorations (learning [17])
MPEG4 (30 fps)	81	144
H.264 (15 fps)	91	149
mad (22k)	86	149
susan (384x288)	78	135
ispell (largespell)	82	139
FFT (32 fps)	75	119
browser (small)	89	141

approach benefits from reduced number of explorations due to the relationship between current performance and VFS action in (4) when compared with the exploration using a uniform probability distribution in [17] (see Lemma I, Appendix B). The applications FFT and susan were found to have the lowest number of explorations. This is because these applications exhibit less workload variation during the RL algorithm step. Due to less variation the Q-learning visits similar states repeatedly and learns the VFS controls faster. The applications MPEG4, H.264 and browser showed the highest number of explorations due to higher workload variations. Such variations cause higher number of states to be visited during RL algorithm, requiring more number of explorations as expected.

B. Intra-Application Energy Minimization

A number of experiments are carried out with intra-application workload and performance variations showing comparative evaluation of the proposed adaptive approach.

1) *Workload Variations*: Fig. 7 shows the experimental results of an H.264 decoder decoding at 24 VGA fps, used as a case study, highlighting the adaptation to intra-application workload variations. Fig. 7.(a) shows the predicted workload together with the observed mean CPU workload over a moving window of $n=200$ decision epochs, while Fig. 7.(b)-(d) show the resulting average slack ratios caused by RL algorithm and adaptation steps, the Q-table states and the VFS control actions chosen over the decision epochs (in terms of frames). As can be seen, initially the governor starts to learn the VFS control actions (Fig. 7.(d)), which results in performance offset in terms of \mathcal{L} (Fig. 7.(b)). As the governor initiates exploiting some of these learnt VFS controls, \mathcal{L} starts to settle to near-zero values. During this phase the Q-table states vary depending on the predicted workloads, while the VFS actions are chosen from the Q-table (Fig. 7.(c)-(d)).

Note that after the 1271th frame the workload profile changes within the application (observation 1, Section II), which is detected through comparison of the observed mean workload, C_n with the base workload (C_b) of the Q-table (Fig. 4). Due to such variation in the workload, the proposed governor carries out learning transfer from the old Q-table with $C_b=2.5 \times 10^7$ to the new Q-table with $C_b=1.5 \times 10^7$ (Section III-B1). The learning transfer is then followed by further exploration of the Q-table to update the VFS controls to achieve near-zero \mathcal{L} values. Due to such re-exploration, the \mathcal{L} values are perturbed again with over-performance at reduced C_n (Fig. 7.(b)). Since the proposed governor adapts the VFS controls in the presence of intra-application workload variations, it can effectively minimize energy, while meeting the application performance requirement.

Fig. 8 plots the normalized energy and performance values of the proposed energy minimization approach with different

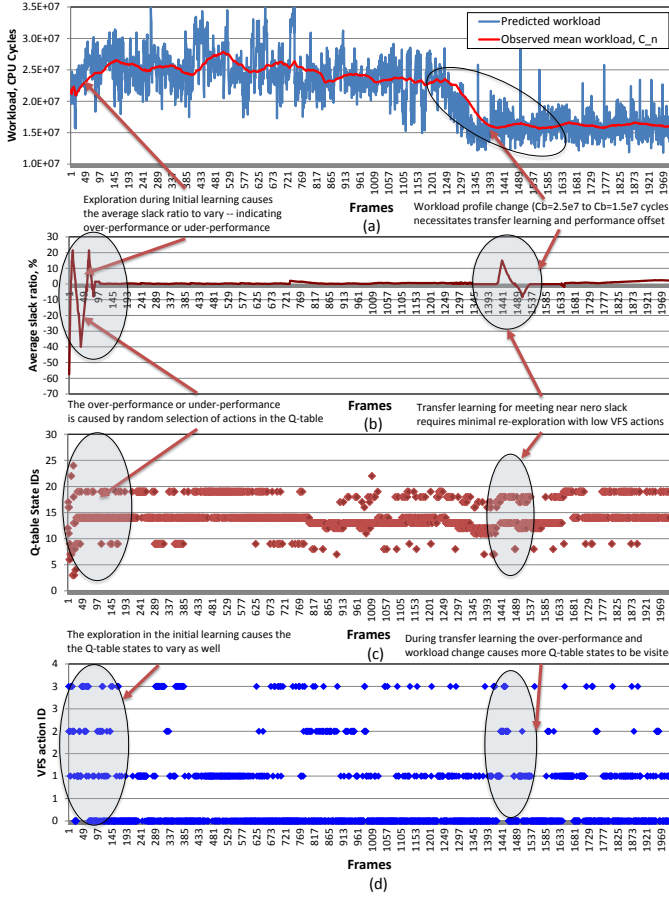


Fig. 7. (a) Predicted and observed mean workloads, (b) average slack ratios (in %), (c) VFS controls, and (d) corresponding Q-table states of an H.264 decoder frame rates and resolutions of H.264, compared with the existing approaches. Normalization is carried out to give comparative figures between different approaches covering the dynamic range of performance and workload variations for various applications. Figures 8.(a)-(c) show the results of decoding QVGA resolution with 15 fps, 24 fps and 29.97 fps, while Figures 8.(d)-(f) show the same for decoding SVGA resolutions. The performance is normalized with respect to the required performance per frame (T_{ref}) and the energy normalization is carried out with respect to Oracle, which is found through offline determination of optimized VFS controls for the observed CPU workloads. With such normalization, normalized performance of lower than 1 means over-performance, higher than 1 means under-performance and close to 1 means on par performance. Similarly, normalized energy of lower than 1 means less energy consumption, higher than 1 means more energy compared to the Oracle; close to 1 means more effective energy minimization, provided the performance is also on par. The normalized energy and performance results are obtained through averaging the decoder results of three different video clips (*football*, *flower* and *foreman*) from *xiph* video repository¹. The results of the proposed approach are compared with Linux's ondemand governor [11], predictive and learning-based approaches. Predictive approach is implemented using [14] without any explicit performance information from the application; for higher predicted workload higher VFS is applied, while for lower predicted workload lower VFS is applied. Learning-based

¹ <http://media.xiph.org/video/derf/>

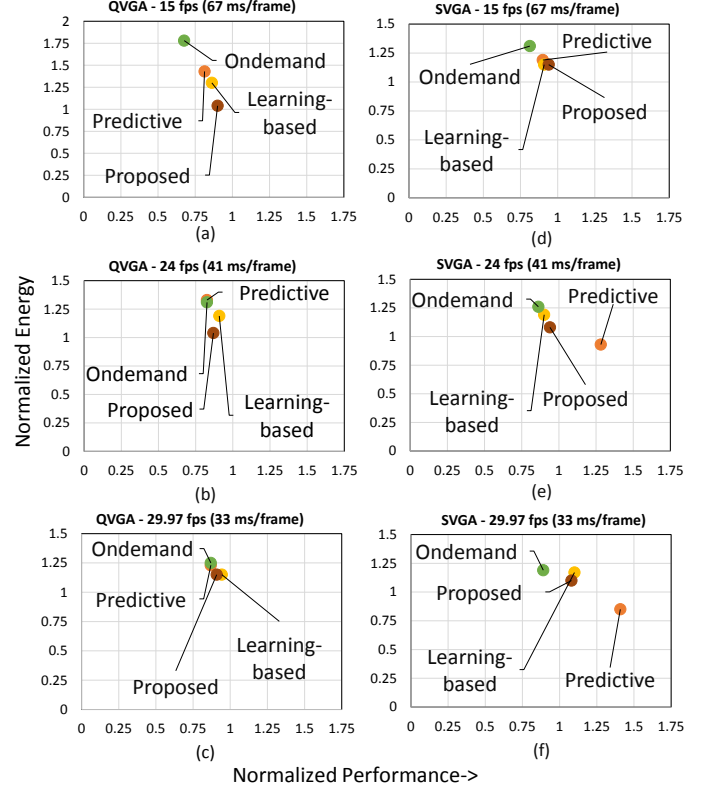


Fig. 8. Comparative evaluation of energy minimization of H.264 video decoder with different performance requirements (normalized performance of 1 means on par performance, > 1 means under-performance and < 1 means over-performance; normalized energy of > 1 means higher energy consumption, < 1 means lower energy consumption; ≈ 1 means effective energy minimization provided that performance is also on par)

approach is implemented using a single Q-table based learning proposed in [17] with application-specific controller, configured with the application performance requirement.

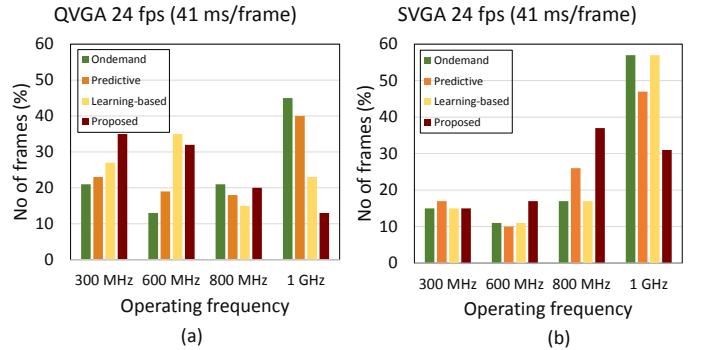


Fig. 9. Comparative histogram of operating frequencies applied in H.264 decoding (a) QVGA, and (b) SVGA frame resolutions, 24 fps each

As can be seen, the ondemand governor consistently over-performs when compared with the other approaches. This is because it does not interact with the applications' performance requirements. As a result, it generates the highest energy consumption among all approaches. The predictive energy minimization approach is also oblivious to applications' performance requirements, and hence it fails to minimize energy consumption effectively meeting the applications' performance requirements. For example, in the case of QVGA the predictive approach over-performs and incurs higher energy consumption (Fig. 8(a)-(c)). However, for SVGA with 24 and 29.97 fps it under-performs, which makes the energy savings achieved

through the predictive approach ineffective (Fig. 8(e)-(f)). The learning-based approach [17] performs better than the ondemand and predictive approaches as it learns the VFS controls based on the performance requirements. However, since it uses a single Q-table formulation it adapts poorly to intra-application workload variations. Our proposed approach can reduce energy consumption through detection of and adaptation to such variations. However, the reduction in energy consumption that can be achieved using the proposed approach depends on the number of intra-application workload variations detected. For example, in the case of Fig. 8.(c)-(f), the proposed approach does not offer much of an energy saving when compared to the learning-based approach [17] due to less workload variations (maximum 1 in the case of decoding SVGA frames at 24 fps). However, in the case of Fig. 8.(a)-(b), there is an energy reduction of up to 20% when compared to the learning-based approach as the number of workload variations are much higher (4 for both cases: decoding QVGA frames at 15 and 24 fps).

To give further insight into the energy minimization of different approaches compared (Fig. 8), Fig. 9 plots the histograms of H.264 decoding QVGA and SVGA resolutions at 24 fps each. As can be seen from Fig. 9.(a), for the low resolution QVGA video, the learning-based, predictive and proposed approaches execute most of the frames at 300MHz or 600MHz. The ondemand, however, executes more than 45% of the frames at 1 GHz due to its CPU utilization-based VFS control. For decoding videos with different resolutions (Fig. 9.(b)), the proposed approach generates a balanced frequency utilization depending on the applications' performance requirements. As a result, it can effectively minimize energy compared to the other approaches.

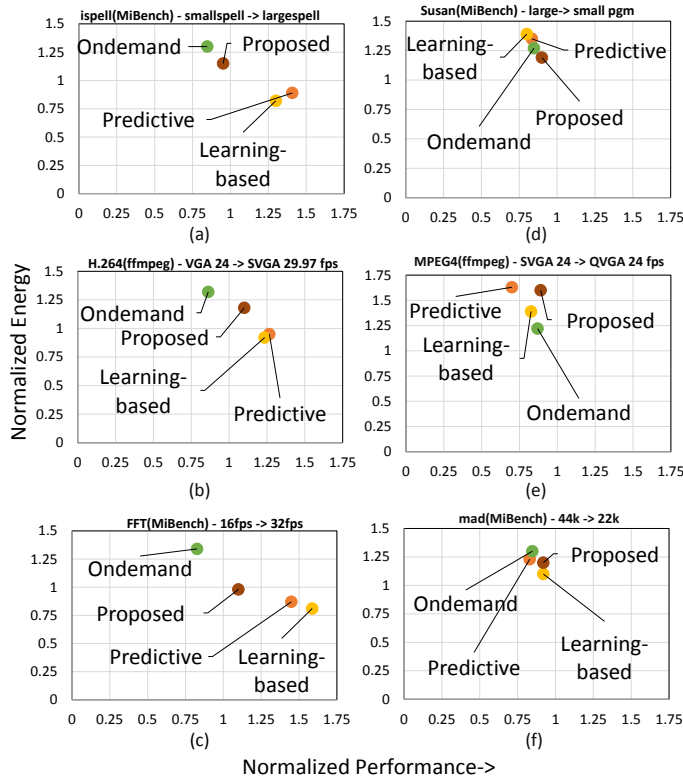


Fig. 10. Comparative energy and performance trade-offs with intra-application performance variations for different benchmark applications (performance and energy consumption values normalized similar to Fig. 8)

2) *Performance Variations*: Fig. 10 depicts the normalized energy and performance values (with respect to Oracle) of

different benchmark applications with varied performances within the applications. Figures 10.(a)-(c) show the variation from a lower performance to a higher performance for ispell (MiBench), H.264 (ffmpeg) and FFT (MiBench) applications, while Figures 10.(d)-(f) show the variation from a higher performance to a lower performance for susan (MiBench), MPEG4 (ffmpeg) and mad (MiBench) applications. The normalized energy and performance results are obtained through averaging three observations, each with input sequence of 3000 decision epochs (i.e. frames for MPEG4, H.264, FFT and susan, spelling task for ispell and audio packet for mad).

As can be seen, when the performance requirement increases from low to high, both predictive and learning-based approaches under-perform (Fig. 10.(a)-(c)). On the other hand, when the performance requirement decreases, these approaches over-perform and incur higher energy consumptions (Fig. 10.(d)-(f)). This is because both approaches cannot adapt to performance variations due to lack of interactions between the layers. The ondemand governor, however, shows trends of scaling with the performance requirements, but it consistently over-performs. The proposed approach can effectively scale with the variation in the performance requirement and adapt through learning transfer (Section III-B2). As a result, it shows effective energy minimization across all experiments saving on average 33% and 24% when compared with the predictive and learning-based approaches in the case of MPEG4 performance variation (Fig. 10.(e)) and 30% when compared with the ondemand governor in the case of FFT performance variation (Fig. 10.(c)).

C. Inter-Application Energy Minimization

Fig. 11 plots the comparative normalized energy consumptions of energy minimization approaches for three inter-application scenarios. Fig. 11.(a) shows the switch from MPEG4 decoding 24 VGA fps to H.264 decoding 15 VGA fps (i.e. high performance to low performance inter-application switch), Fig. 11.(b) shows the switch from ispell with small text task to mad 44k audio packets (i.e. high performance to low performance switch), Fig. 11.(c) shows the switch from susan large (image size of 384x288 pixels) to FFT 32 wave fps (high performance to high performance), Fig. 11.(d) shows the switch from browser (small pages) image to MPEG4 decoding 30 fps (low performance to high performance), Fig. 11.(e) shows the switch from mad (44k) audio decoder to susan large (high performance to high performance), and finally, Fig. 11.(f) shows the switch from FFT (16 fps) to browser medium sized pages (page size upto 35k, demonstrating low performance to high performance). For each inter-application variation, two different switching intervals of 1000 and 2000 decision epochs are used. The energy values are normalized with respect to the proposed approach.

From Fig. 11 two observations can be made. First observation is related to inter-application energy minimization for a given switching interval; as can be seen, for an application switch from a higher performance to lower performance requirement the proposed approach saves up to 38% and 22% for switching interval of 2000 decision epochs compared to the predictive and learning-based approaches, respectively (Fig. 11.(a)). However, when the application switches from a lower to higher performance requirement the proposed approach consumes more energy, while meeting the new application performance requirement when compared with the predictive and learning-based approaches.

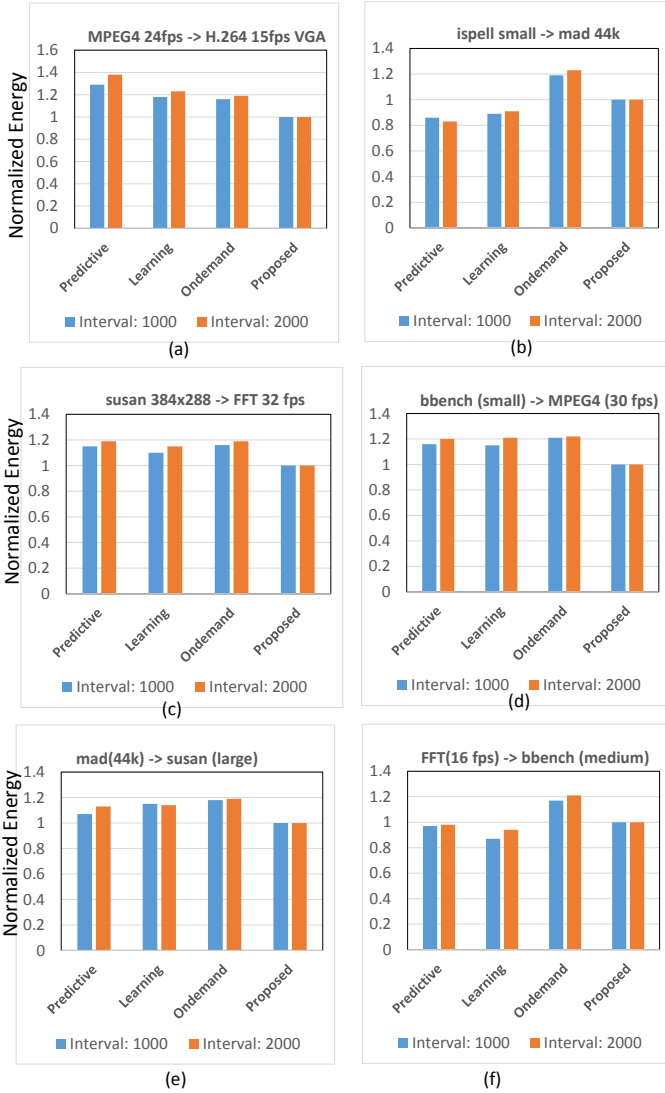


Fig. 11. Comparative normalized energy with inter-application variations

This is because the proposed adaptive approach adapts to higher VFS control actions to meet the increased performance requirement. Both predictive and learning-based approaches fail to meet the application performance requirement despite their energy savings. Similar adaptations are also observed for the other inter-application switches. As expected, the ondemand consistently over-performs compared to the proposed approach. The second observation is related to the change of switching interval; as can be seen with higher switching interval the proposed approach exploits the learning and adaptation for longer time. This leads to higher energy savings compared with the other approaches. For example, for a change of switching interval from 1000 to 2000, the proposed approach saves up to 6% more energy compared to the predictive approach.

V. LEARNING OVERHEADS

Adaptative energy minimization in the proposed approach is achieved through inter-layer interactions and learning algorithms that have the following two impacts. Firstly, when exploring during the RL or the learning transfer, the under-performance can cause real-time deadlines to be missed. Secondly, due to additional computation and storage required by the learning algorithms (Section III) overheads (time and energy) are also

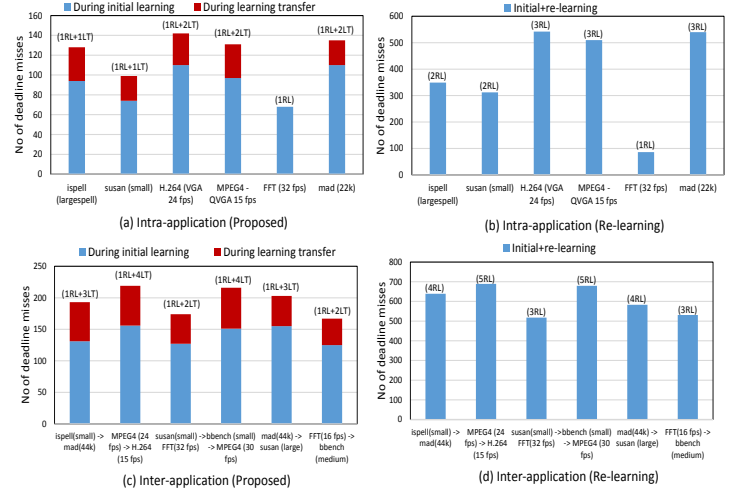


Fig. 12. Worst-case number of deadline misses for (a) intra-application variations using the proposed approach, (b) intra-application variations using the learning approach [17] applied to each variation, (c) inter-application variations using the proposed approach, and (d) inter-application variations using the learning approach [17] applied to each variation

caused per decision epoch. To demonstrate the impact of learning through real-time deadline misses and overhead, Fig. 12.(a)-(b) show the worst-case number of deadline misses for intra-application scenarios, while Fig. 12.(c)-(d) show the same for inter-application scenarios using the proposed and learning [17] approaches, respectively. For all application scenarios, the worst-case number of deadline misses was recorded from five consecutive runs using the input sizes specified, each application with 3000 decision epochs. For the intra-application scenarios (Fig. 12.(a)-(b)), the following observation can be made. As can be seen, the number of worst-case deadline misses depends on the intra-application workload variations and the initial random explorations. The mad, MPEG4 and H.264 applications exhibit the highest number of deadline misses as these applications go through one initial RL and two intermediate learning transfers (LTs) each. The other applications, such as ispell and susan only go through one initial RL and one LT, while FFT goes through one initial RL without any LT. As a result, the number of deadline misses in these applications are lower. Similar observations can also be made from the inter-application scenarios in Fig. 12.(c)-(d), as the worst-case number of deadline misses depend on the number of workload and performance variations encountered. As can be seen, the inter-application scenarios with MPEG4 (24 fps) to H.264 (15 fps) and browser (small) to MPEG4 (30 fps) incur higher number of deadline misses as they go through one initial RL and four LTs. On the other hand, the scenarios with susan (small) to FFT (32 fps) and FFT (16 fps) to browser (medium) incur lower number of deadline misses as these go through one initial RL and two LTs each. It is to be noted that over the 3000 decision epoch of the observation period for each application, the worst-case number of deadline misses accounts to only 4.8% for the intra-application scenarios and 3.8% for the inter-application scenarios using the proposed adaptive approach. When the learning approach [17] is used for adapting to the workload or performance variations, the worst-case number of deadline misses increases to 18% due to inter-application variations and 13% in the presence of inter-application variations. This increase in overheads is caused by increased number of explorations during re-learning

as opposed to learning transfer-based explorations, as explained in Proposition I (see Appendix B).

To demonstrate the impact of learning due to additional computation and storage, Fig. 13 plots the average learning overheads of the proposed approach per decision epoch, compared with the existing approaches: ondemand [11], learning [17] and predictive [14]. The time overheads are evaluated by averaging

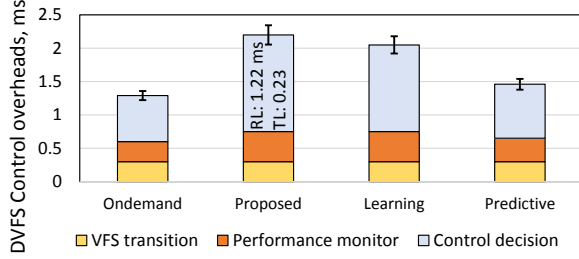


Fig. 13. Comparative time overheads ($TOVH$) of different approaches the differences of per frame execution times of a *ffmpeg* video decoder decoding three sequences ($T_{ref} = 31ms$) with and without energy minimization approaches. As can be seen, the measured overheads have the following three component delays:

- 1) *VFS Transition Delay*: it is a less variable delay due to transition of CPU frequencies. ARM Cortex A8 has a transition delay of about $300 \mu s$ ($0.3 ms$) [30].
- 2) *Performance Monitor Delay*: This delay includes the time taken by reading the system clock for evaluating the performance (typically few microseconds to up to $0.3ms$ depending on the number of times the clock is read [31]) and performance counter registers' access (each access has the typical measured delay of $\approx 20 \mu s$, including the overheads of the C-wrapped assembly instructions).
- 3) *Control Decision Delay*: This delay includes the time taken by the various control steps and varies/depends on the complexity of the control.

As expected, the control decision dominates the overheads in all the approaches as these require number of computation steps (such as reinforcement and learning transfer algorithms with multiple fixed point calculations). The proposed approach exhibits the highest time overhead of $2.1ms$, with up to 8% deviation of the control decision time depending on the random explorations during initial RL and intermediate LTs. Compared with the learning approach [17], the proposed approach uses additional interactions between the layers and learning transfer-based adaptation steps to minimize energy further in the presence of intra- and inter-application variations, with up to $1.22 ms$ and $0.23 ms$ respectively for the RL and TL steps. The predictive and ondemand approaches have lower overheads due to simpler control decisions and less performance counters' access. The higher overhead of the proposed approach highlights one of the limitations of the proposed approach. However, such overhead is higher compared to application performance requirement, the T_{ref} can be re-defined as multiple of decision epochs to effectively minimize energy consumption.

To demonstrate the impact of learning choices made in terms of size of the Q-tables in the RL algorithm, Fig. 14 plots the average energy (in %) and time overheads (in ms) for different Q-table sizes with the following state-action entries: 15 states and 4 actions (i.e. 15×4), 30 states and 4 actions (i.e. 30×4) and 40 states and 4 actions (i.e. 40×4). The energy overheads are evaluated by comparing the average energy

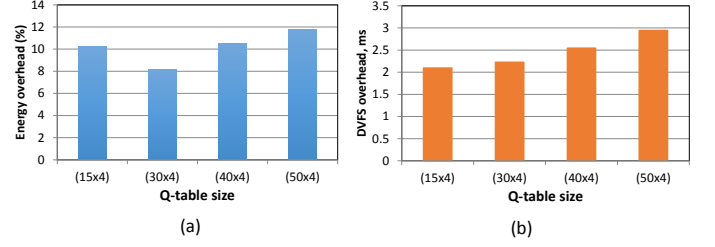


Fig. 14. (a) Energy, and (b) time overheads for different Q-table sizes

consumptions of the proposed approach with offline profile-generated energy consumption in Oracle for similar performance levels (Fig. 14.(a)), while time overheads are measured by observing the CPU times elapsed during learning and VFS action ($TOVH$) averaged per decision epoch (Fig. 14.(b)). Both measurements are obtained through two *ffmpeg* (H.264 and MPEG4) and four MiBench (FFT, susan, ispell and mad) benchmark applications, running over 3000 decision epochs each with the corresponding input sequences. As can be seen, the energy overheads increase slightly with the increased Q-table sizes (Fig. 14.(a)). This is because of the following two reasons. Firstly, with higher number of states, the Q-tables now have higher complexity and require longer exploration times, which results in slower convergence over time. The impact of this can be observed in Fig. 14.(b), as the time overhead per decision epoch marginally increases with increased Q-table sizes. Secondly, due to increased time overheads, the effective deadline per decision epoch ($T_{ref} - TOVH$) reduces, which requires slightly higher VFS control to be applied to meet the performance requirements, resulting in higher energy consumption. It can be noted that for the lowest Q-table size of (15×4) , the proposed approach also incurs slightly higher energy consumption. Due to lower workload variations covered with in the Q-table, the proposed approach goes through more LTs, requiring further explorations (see Fig. 15).

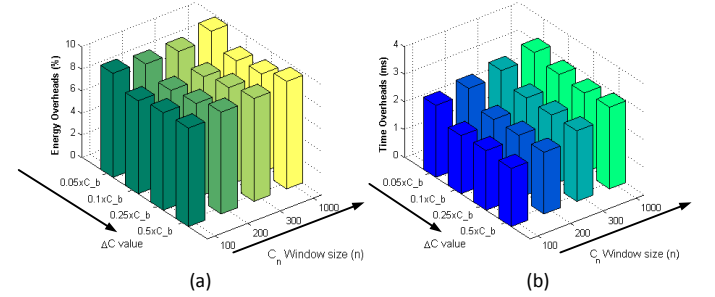


Fig. 15. (a) Energy, and (b) time overheads for varying workload bins (ΔC)

To investigate into the impact workload bin (ΔC) and window sizes of average workload (C_n) computation, Fig. 15(a) and (b) show the 3D bar plots of the energy and time overheads incurred by the proposed approach for the following ΔC values: $0.05 \times C_b$, $0.1 \times C_b$, $0.2 \times C_b$ and $0.25 \times C_b$. For each ΔC value, the window size n is varied between 100, 200, 300 and 1000 decision epochs. The energy and time overheads are evaluated repeating the experiments reported in Fig. 14 with the Q-table size of (30×4) . Fig. 15(a)-(b) demonstrate the energy consumption and time overhead trade-offs with ΔC values for a given moving average window size. As can be seen, at the lower ΔC values, the energy and time overheads are higher. This is because the workload variations covered by the Q-table is lower, which increases the number of LTs needed to adapt

to workload variations for a given application. When the ΔC values are higher, coarser workload variations are covered by the Q-table, which gradually reduce the learning time overheads due to less number of LTs. However, at increased ΔC values, the normalized energy overhead increases marginally. This is because coarser workload bins cause loss of precision of VFS controls needed for effective energy minimization.

The moving average window size for C_n also demonstrates energy and time overheads trade-offs (Fig. 15(a)-(b)). At lower window size, the proposed approach experiences higher workload variations in the short-term, causing an increase in the LTs needed to adapt to workload variations. This causes the energy and time overheads to increase slightly. As the window size is gradually increased, the number of intra-application variations decrease, reducing the energy and time overheads. However, when the window size is too large, it causes higher energy and time overheads due to the following two reasons. Firstly, such large window size poorly represents workload variations through C_n , which causes loss of control precision. Secondly, due to such large window size the computation and storage during runtime also increases, which directly increases the computation time overheads. Such increased time overheads, in turn, reduces the opportunity to reduce energy effectively as demonstrated by the highest energy overheads ($\approx 10\%$) for a given ΔC value of $0.05 \times C_b$ (Fig. 15(a)). The best trade-off between energy and time overheads is obtained at ΔC value $= 0.1 \times C_b$ and C_n window size of 200.

VI. SCALING TO MULTI-CORE SYSTEMS

The proposed approach is also implemented and validated in multi-core systems. The implementation requires simple modifications to the approach presented in Section III. First, the predicted workload per core is normalized with respect to the total system workload as

$$\bar{C}_{t+1}^j = \frac{\hat{C}_{t+1}^j}{\sum_j \hat{C}_{t+1}^j}, \quad (13)$$

where \hat{C}_{t+1}^j is the predicted workload in CPU cycles and \bar{C}_{t+1}^j is the normalized workload for the j -th processor core ($j=1$ to J , J is the total number of cores in the system). With the given normalized workload (\bar{C}_{t+1}^j) and the average slack ratio (\mathcal{L}_t) bins a number of Q-table states are defined and organized in rows (similar to (2) and (3)). For each state, the available VFS control options are used in the action space organized in the columns to form the Q-table. This Q-table is then shared among the processor cores to allow reinforcement learning through one core action update per decision epoch (controlled in a round robin fashion). Such VFS control per decision epoch has two distinct advantages: (a) automatic learning transfer between cores when similar workload is predicted (see Fig. 17), and (b) Q-table complexity is reduced significantly as opposed to controlling multiple cores per decision epoch, which requires combinations of VFS controls of all cores in the Q-table [29]. To ensure that the Q-table accesses are mutually exclusive, mutex-based Q-table access is implemented in the governor (see Appendix A). After the initial learning when an intra- or inter-application workload or performance variation is detected, learning transfer is carried out as shown in Section III-B.

To validate the effectiveness of the approach in multi-core systems, further experiments are carried out on the following:

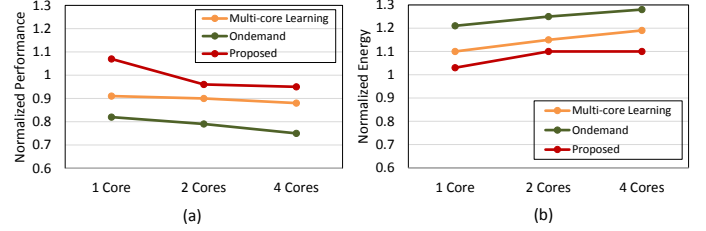


Fig. 16. Comparative (a) normalized performance, and (b) normalized energy consumptions of different approaches for varying number of processor cores

a Xilinx Zync ZC702 SoC [26] with two ARM Cortex-A9 cores, and a Hardekernel Odroid-XU SoC [28] with four ARM cores (using cluster switching between Cortex A-15 and Cortex A-7 cores). The Zync SoC supports three VFS control points: 666MHz at 1V, 333MHz at 0.8V and 222MHz at 0.75V and the Odroid-XU SoC has six VFS control points: 1.6GHz at 1.2V, 1.0GHz at 1V, 600MHz at 0.8V and 300MHz at .75V. Since performance counters are limited on these systems, observed CPU workloads are obtained by dividing the CPU utilization by the product of time elapsed and last operating frequency on the CPU. An H.264 based video decoder application is executed with a football sequence of approximately 3000 frames using the following three approaches: multi-core DVFS control approach [29], Linux ondemand governor per core [11] and the proposed approach. The approach [29] was chosen for comparison as it is the closest match using reinforcement learning based DVFS controls in multiprocessor systems. However, for equivalence and comparability between [29] and our approach the thermal constraint was not used in [29]. Fig. 16(a) shows the normalized performance, while Fig. 16(b) plots the normalized energy consumptions of the approaches. Similar to Fig. 8, the performance is normalized with respect to the required performance per frame (T_{ref}) and the energy normalization is carried out with respect to the Oracle.

From Fig. 16(a), it can be seen that the proposed approach continues to provide with similar performance as the Oracle. The multi-core learning [29] and ondemand [11] approaches over-perform as these approaches cannot learn and adapt to application and performance variations effectively. Due to their higher performance, these approaches experience (up to 18%) higher energy compared to the proposed approach (Fig. 16(b)).

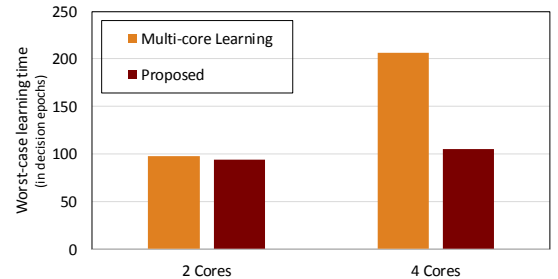


Fig. 17. Comparative worst-case learning overheads for multi-core systems

Using such learning transfer-based approach to multi-core systems has the added advantage of controlled learning overheads as shown in Fig. 17. As can be seen, the proposed approach continues to learn effectively using similar number of learning time as the single core (see Table III) in terms of the number of decision epochs taken in the worst-case. This is because, each core carries out exploration of the DVFS controls in

each decision epoch using its normalized predicted workload and the overall application performance in the state space pair, which make the learning evaluations between cores automatically transferrable. However, when DVFS control of multiple cores are considered in each decision epoch, the Q-table requires using combinations of all DVFS options. Due to such large action space, the worst-case learning time increases by more than 2X when the number of cores increases from 2 to 4.

VII. CONCLUSIONS

An adaptive energy minimization approach for embedded systems has been proposed, capable of adjusting to workload and performance variations within and across applications. The energy minimization is enabled through reinforcement learning algorithm for identifying the suitable VFS controls based on predicted workloads for a given application performance requirement. To ensure VFS controls are adjusted when workloads or performance variations take place learning transfer based adaptation is carried out, guided by the feedback from the CPU performance counters. The proposed approach is implemented as a power governor in Linux OS and extensively validated through experiments using different benchmark applications and number of cores. The approach is expected to provide effective energy minimization for current and future generations of embedded systems that typically execute multiple applications.

REFERENCES

- [1] D. Flynn. An ARM Perspective on Addressing Low-power Energy-efficient SoC Designs. in *Proc. of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*, pp.73–78, 2012.
- [2] Y. Tan, W. Liu, and Q. Qiu. Adaptive power management using reinforcement learning. in *Proc. of Intl. Conference on Computer-Aided Design, ICCAD*, New York, NY, USA: ACM, 2009, pp.461–467.
- [3] J. Pouwelse, K. Langendoen and H.J. Sips. Application-directed voltage scaling. in *IEEE Trans. Very Large Scale Integration Systems (TVLSI)*, vol.11, no.5, pp.812–826, Oct. 2003.
- [4] L. Benini, A. Bogliolo, and G. De Micheli, “A survey of design techniques for system-level dynamic power management,” in *IEEE TVLSI*, vol.8, no.3, pp.299–316, June. 2000.
- [5] K. Choi, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for an MPEG Player. *Journal of Low Power Electronics*, vol. 1, no. 1, pp.27–43, Apr. 2005.
- [6] W. Yuan and K. Nahrstedt. Practical voltage scaling for mobile multimedia devices. in *Proc. of the 12th Annual ACM International Conference on Multimedia*, ACM, pp.924–931, NY, USA, 2004.
- [7] Y. Gu and S. Chakraborty. Control theory-based DVS for interactive 3D games. in *Proc. of the 45th Annual Conference on Design Automation (DAC)*, New York, USA: ACM Press, 2008, pp.740–745.
- [8] S. Sinha, J. Suh, B. Bakaloglu and Y. Cao. Workload-aware neuromorphic design of the power controller. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 3, pp.381–390, Sep. 2011.
- [9] G. Dhiman and T.S. Rosing. System-level power management using online learning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 5, pp.676–689, May. 2009.
- [10] M. Pedram. Power optimization and management in embedded systems. in *Proc. of the Asia and South Pacific Design Automation Conference, ASP-DAC'01*, ACM, pp.239–244, Yokohama, Japan, 2001.
- [11] V. Pallipadi and A. Starikovskiy. The Ondemand governor. in *Proc. of the Linux Symposium*, 2006.
- [12] R. Jejurikar and R. Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems *Proc. of ISLPED'04*, pp.78–81, August, 2004.
- [13] H. Jung and M. Pedram. Continuous frequency adjustment technique based on dynamic workload prediction. in *21st IEEE Intl. Conference on VLSI Design, VLSID*, IEEE, 2008, pp.249–254.
- [14] K. Choi, R. Soma and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. in *Proc. of ISLPED'04*, 2004, pp.174–179.
- [15] S. Yue, D. Zhu, Y. Wang, and M. Pedram. Reinforcement learning based dynamic power management with a hybrid power supply. in *IEEE 30th Intl. Conference on Computer Design (ICCD)*, 2012, pp.81–86.
- [16] A.K. Das, R.A. Shafik, G.V. Merrett, B.M. Al-Hashimi, A. Kumar and B. Veeravalli. Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems. in *Proc. of DAC'14*, pp.1–6, June, 2014.
- [17] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu. Achieving autonomous power management using reinforcement learning. *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 2, pp.24:1–24:32, Apr. 2013.
- [18] R. Ye, Q. Xu. Learning-based power management for multi-core processors via idle period manipulation. In *IEEE TCAD*, vol.33, no.7, pp.1043–1055, 2014.
- [19] BeagleBoard. BeagleBoard-xM Rev C System Reference Manual, 2010. [Online]. Available: <http://beagleboard.org>
- [20] FFmpeg A complete, cross-platform solution to record, convert and stream audio and video. [Online]. Available: <https://www.ffmpeg.org/>
- [21] M.R. Guthaus, M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. in *IEEE International Workshop on Workload Characterization (WWC)*, IEEE, pp.3–14, Dec., 2001. [Online]. Available: <http://www.eecs.umich.edu/mibench/>
- [22] FFmpeg Open-source Mozilla Firefox browser. [Online]. Available: <https://www.mozilla.org/>
- [23] BBench Browser benchmarking tool. [Online]. Available: <http://bbench.eecs.umich.edu/>
- [24] T. Jiang, D. Grace and P.D. Mitchell Efficient exploration in reinforcement learning-based cognitive radio spectrum sharing *IET Communications*, vol. 5, Iss. 10, pp. 1309–1317, 2010.
- [25] O.S. Unsal, and I. Koren. System-level power-aware design techniques in real-time systems in *Proc. of the IEEE*, vol.91, no.7, pp.1055–1069, July, 2003.
- [26] Xilinx Inc. Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.2) 2012.
- [27] W. Yuan, K. Nahrstedt, S. Adve, D.L. Jones and R.H. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. in *Proc. SPIE 5019, Multimedia Computing and Networking*, pp.1–13, Jan, 2003.
- [28] Hardkernel. Odroid-XU by Hardkernel. [Online]. Available: <http://www.hardkernel.com>. Last Accessed 10 Dec. 2014.
- [29] Y. Ge and Q. Qiu. Dynamic Thermal Management for Multimedia Applications Using Machine Learning. in *Proc. of the 48th Design Automation Conference (DAC)*, New York, USA, pp.95–100, 2011.
- [30] S. Sangyoung, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram and N. Chang. Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors. in *IEEE TCAD*, vol.32, no.5, pp.695–708, May, 2013.
- [31] ARM. Cortex-A8 Technical Reference Manual. [Online]. Available: <http://www.arm.com>. Last Accessed 7 April 2015.

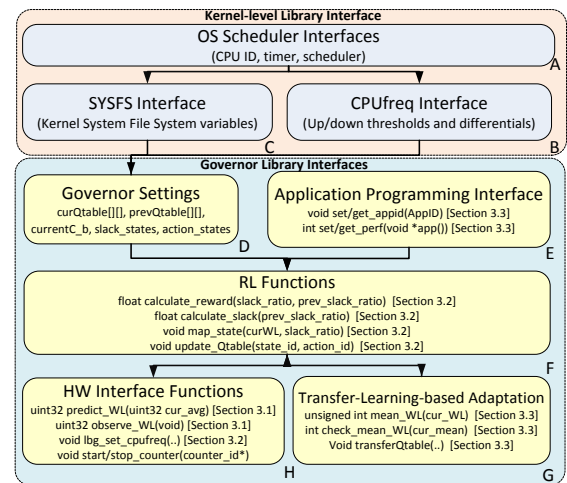


Fig. 18. Implementation of the proposed approach showing different interfaces

APPENDIX A: LINUX GOVERNOR IMPLEMENTATION

Fig. 18 shows the block diagram of the Linux power governor implementation consisting of the following interfaces:

A. Kernel-level Library Interfaces

These interfaces comprise of the scheduler, *CPUfreq* and *SYSFS* interfaces. The scheduler interface (block A) defines the CPU IDs, system timer and functions to establish governor control of a specific CPU. The *CPUfreq* interface (block B) retrieves information related to processor frequency steps. The *SYSFS* interface (block C) defines file system variables/constants.

B. Governor Library Interfaces

These interfaces include the following:

1) *Governor settings (block D)*: define the Q-tables (*curQTable* and *prevQTable*), each with its base workload C_b , performance requirement T_{ref} and the state-action pairs.

2) *Application programming interface (block E)*: sets up the inter-layer interactions between the application and runtime layers and enables the detection of intra- or inter-application performance variations. It is implemented through a thread-safe handshake signal (called *rts→handshake*) between application and runtime system, and a *SYSFS* variable storing the current application performance requirement. The kernel initially starts a notification process with this signal, which is updated and notified by the application layer at each decision epoch. This notification is then followed by *set_perf(..)* function to set the performance requirement (T_{ref}) in the *SYSFS* variable.

3) *Hardware interface (block H)*: defines the interaction between the runtime and hardware layers and enables the detection of workload variations within and across applications. This is carried out through observing the average workload from the CPU performance counters. The interface also defines workload prediction and observation functions *predict_WL(..)* and *observe_WL(..)*. During observation of the workload, *start_counter(..)* and *stop_counter(..)* IO functions are used to start and stop the CPU performance counters using the *ioctl* interface in Linux. The frequency is then set for the system using *lbg_set_cpufreq(..)* function.

4) *RL functions (block F)*: set up the learning functions in the Q-table (Section III-A2). Based on the predicted workload, *map_state(..)* maps the system's current state, the *calculate_reward(..)* function calculates reward of a selected VFS action. The calculated reward is then updated through *update_Qtable(..)* function using the Q-values given by (5).

5) *Learning transfer-based adaptation (block G)*: is implemented through a set of functions. For detecting workload variations, the short-term mean workload is calculated using *mean_WL(..)* and for performance variations are communicated through API. When a variation is detected, the Q-table is updated and learning is transferred with a new base workload value (C_b) using *transferQtable(..)*. A total of eight C_b are pre-defined for Q-table update and learning transfer.

The above kernel-and user-level functions and interfaces were implemented as Linux governor module within *CPUfreq*. Since floating point calculations are limited in kernel-level, appropriate scaling was applied to different variables and constants discussed in Section III. The governor can be applied in the Linux command-line through *CPUfreq* utility as:

```
cpufreq_set --cpu 0 --governor tlb
```

The *cpufreq_set* is a command line utility to administer the *CPUfreq* governor settings; the *-cpu* option is followed by the CPU ID (e.g. '0') and the *-governor* option specifies the governor name, (e.g. 'tlbg': transfer learning-based governor).

APPENDIX B

LEMMA I: For exploration of all $|\mathcal{S}|$ states in a Q-table, each with $|\mathcal{A}|$ actions, the minimum number of explorations required by an exponential probability distribution-based exploration is given by: $(\frac{3}{5}|\mathcal{S}||\mathcal{A}|)$, which is 40% less than that required by an uniform probability distribution-based exploration: $(|\mathcal{S}||\mathcal{A}|)$.

Proof: The exploration of Q-table states can be divided in two categories considering the slack distribution in the Q-table (given by (2)): exploration of the states with near-zero (i.e. $\pm 5\%$ slack values) and exploration of the other slack states. The former category consists of $\frac{1}{5}$ th of $|\mathcal{S}|$ and requires exploration of all $|\mathcal{A}|$ actions in the Q-table due to (4). The total number of explorations required by the states in this category amounts to $(\frac{1}{5}|\mathcal{S}||\mathcal{A}|)$. The later category consists of $\frac{4}{5}$ -th of $|\mathcal{S}|$ and minimally requires exploration of only half of the $|\mathcal{A}|$ actions in the Q-table due to relationship between their slacks and actions given by (4). Hence, the total number of explorations required by the states in the this category sums up to $(\frac{4}{5}\frac{1}{2}|\mathcal{S}||\mathcal{A}|) = (\frac{2}{5}|\mathcal{S}||\mathcal{A}|)$. As a result, the minimum number of explorations required for $|\mathcal{S}|$ states by an exponential probability distribution-based exploration is given by: $(\frac{3}{5}|\mathcal{S}||\mathcal{A}|)$. This proves the first part.

The exploration of $|\mathcal{S}|$ states using uniform probability distribution does not exploit the slack and action relationship described in (4). As a result, all slack states with their action states need to be explored, requiring a total of $(|\mathcal{S}||\mathcal{A}|)$ explorations. This can be reduced by 40% using the exponential distribution-based exploration. This proves the second part.

PROPOSITION I: (a) At higher slack states (i.e. $|\mathcal{L}_t| > 5\%$), the learning transfer algorithm will retain the performance and VFS action relationships for most of the states.

(b) If $x\%$ of the higher slack states (i.e. $15\% > |\mathcal{L}_t| > 5\%$) retain their relationships with the chosen actions, the learning transfer algorithm (Algorithm 1) will require minimum $[(\frac{1}{5}|\mathcal{S}||\mathcal{A}|) + (\frac{2}{5}(1-x\%)|\mathcal{S}||\mathcal{A}|)]$ further explorations.

Proof: At higher slack states (i.e. $15\% > |\mathcal{L}_t| > 5\%$), the choice of actions is governed by the current performance level. If the system is currently over-performing, a lower frequency is chosen; however, if the system is currently under-performing, a higher frequency is chosen. When workload of performance varies within and across applications, these relationships are still governed by the current performance level in terms of average slack ratio. The changed workload or performance requirement due to such variation does not affect such relationship (see Section III-B). Hence, most of the states retain the performance and VFS action relationships. This proves part (a).

From Lemma I, the minimum number of explorations required for the lower slack states (i.e. $\pm 5\%$ slack values) is given by $(\frac{1}{5}|\mathcal{S}||\mathcal{A}|)$. When $x\%$ of the higher slack states retain their relationships with the scaled actions, the minimum number of explorations required by the learning transfer algorithm is given by the fraction of remaining higher slack states by half of the total number of possible actions, i.e. $(\frac{2}{5}(1-x\%)|\mathcal{S}||\mathcal{A}|)$ (Lemma I). For all slack states, the learning transfer algorithm (Algorithm 1) requires minimum $[(\frac{1}{5}|\mathcal{S}||\mathcal{A}|) + (\frac{2}{5}(1-x\%)|\mathcal{S}||\mathcal{A}|)]$ further explorations to expedite learning. This proves part (b).