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,

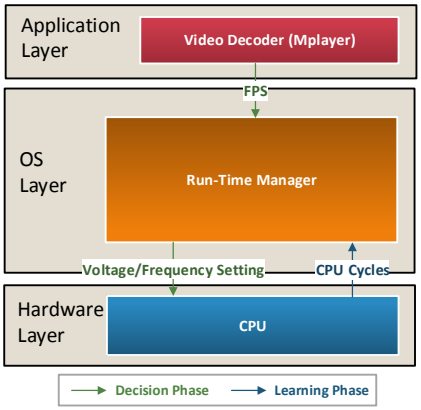

Fig. 1. Run-Time Management system for a video decoder application: a cross-layer approach

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 algorithm is further explained in Section IV-E2. 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.

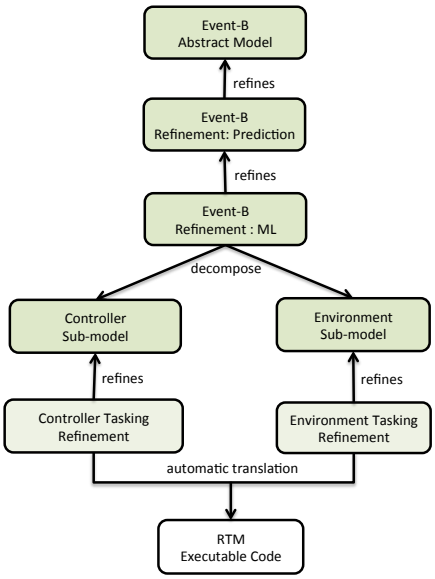

Fig. 2. Event-B formal design of the Run-Time Management system for a video decoder application

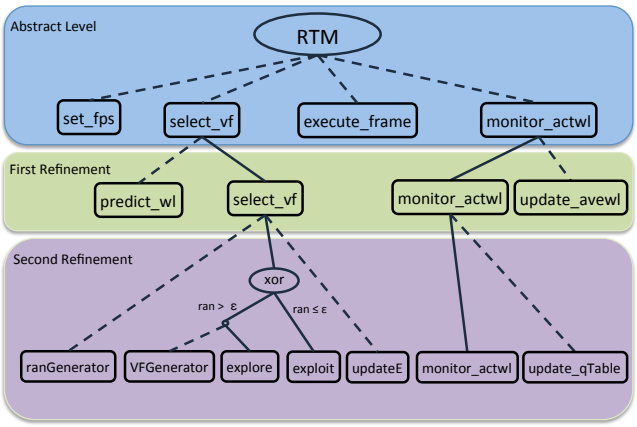

Fig. 3. Event refinement structure of the Run-Time Management system for a video decoder application

In the top level, the value of the V-F is decided nondeterministically 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 nondeterministically assigned to a value from the set *VF*.

Event *select_vf* $\widehat{=}$
begin
act1 : $freq :\in VF$
end

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 concerete *select_vf* event is refining the abstract *select_vf*. The description of these events are as below:

Event *predict_wl* $\widehat{=}$
begin
act1 : $pwl := predict(avgwl)$
end

Event *select_vf* $\widehat{=}$
refines *select_vf*
begin
act1 : $freq := pwl * fps$
end

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 epoch[1], 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 decisions[2]. 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.

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

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

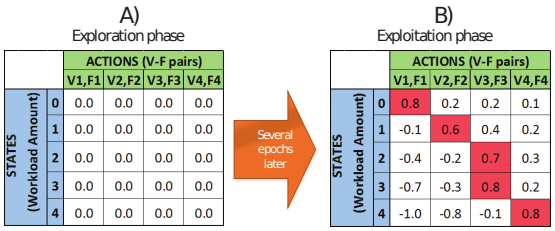

Fig. 4. Q-Table during A) exploration and B) exploitation phases. The red boxes represent the best Action for each State.

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* $\widehat{=}$
refines *select_vf*
when
grd : $random > epsilon$
begin
act1 : $freq := randomVF$
end

Event *exploit* $\widehat{=}$
refines *select_vf*
when
grd : $random \leq epsilon$
begin
act1 : $freq := optimalVF(QTable, pwl)$
end

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 ProB [3] 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:

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

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

[3]The 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 task 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.

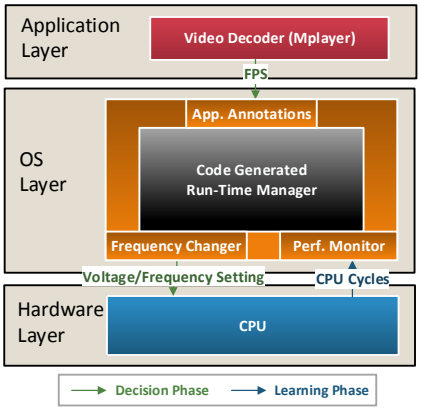

Fig. 5. Code Generated RTM for Video decoding

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* $\widehat{=}$
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

: $lambda \in \mathbb{Z}$
: $avgWeight \in \mathbb{Z}$
: $nextWeight \in \mathbb{Z}$

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

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

Event *costFun_reward_assign* $\widehat{=}$
refines *costFun_reward_assign*
when

grd : $(actwl/freq) \leq dl$

then

act : $costFun\_reward\_value := min(1, costFun\_reward(actwl, freq, dl))$

The *costFun_reward_assign* can 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 distribution[4]. For the video decoder case study, we used FFMPEG[5] libraries, running H.264 video[6] 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 16μJ. 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.

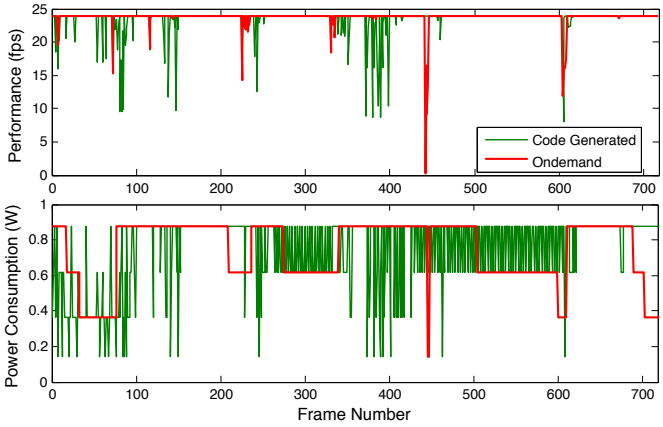

Fig. 6. Performance and Power Consumption of the generated RTM Governor for an H.264 Video

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.

[4]Nelson: https://eewiki.net/display/linuxonarm/BeagleBoard
[5]FFMPEG: http://www.ffmpeg.org
[6]Big 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.