A Model-Based Framework for Software Portability and Verification in Embedded Power Management Systems

Asieh Salehi Fathabadi, Michael J Butler, Sheng Yang,
Luis Alfonso Maeda-Nunez, James Bantock, Bashir M Al-Hashimi,
and Geoff V Merrett

Electronics and Computer Science
University of Southampton
{*asf08r, mjb, sy2u12, lm15g10, jrbb1g11, bmah, gvm*}*@ecs.soton.ac.uk*

Abstract

Run-Time Management (RTM) systems are used in embedded systems to dynamically adapt hardware performance to minimise energy consumption. A significant challenge is that RTM software can require laborious manual adjustment across different hardware platforms due to the diversity of architecture characteristics. Model-driven development offers the potential to simplify the management of platform diversity by shifting the focus away from hand-written platform-specific code to platform-independent models from which platform-specific implementations are automatically generated. Furthermore, the use of formal verification provides the means to ensure that implementations are correct-by-construction. In this paper, we present a framework for automatic generation of RTM implementations from platform-independent formal models. The methodology in designing the RTM systems uses a high-level mathematical language, Event-B, which can describe systems at different abstraction levels. A code generation tool is used to translate platform-independent Event-B RTM models to platform-specific implementations in C. Formal verification is used to ensure correctness of the Event-B models. The portability offered by our methodology is validated by modelling a Reinforcement Learning (RL) based RTM for two embedded applications and generating implementations for three different platforms (ARM Cortex-A8, A7 and A15) that all achieve energy savings on the respective platforms.

Keywords: Run-Time Management, Code Generation, Formal Methods, Verification

1. Introduction

Dynamic Voltage and Frequency Scaling (DVFS) has been used to reduce the energy consumption of mobile and embedded systems at run-time, while maintaining a required Quality of Service (QoS) [1, 2, 3]. In a cross-layer approach to DVFS control, a Run-Time Management (RTM) system interacts with both the application (to ensure that QoS requirements are met) and the hardware platform (to monitor and control core activities). RTM typically includes workload prediction and machine learning algorithms [4, 5].

There are a number of complexities associated with the RTM system implementation. One of the challenges is that it is coupled with the hardware platform specifications, and is implemented individually for each specific platform. Hardware specifications vary from one platform to another, and include a number of characteristic including performance parameters and interaction interfaces. Performance parameters include the range of Voltage and Frequency (VF) settings, the range of workload types to execute an application, and the DVFS latency. Interaction interfaces define the connection between the hardware platform, the application and the RTM. The former influences the RTM algorithms. The later influences the way the RTM interacts with the application and the hardware, e.g. to read QoS from the application, to control VF in the core, and to monitor the workload from the core.

In this paper, we present a framework for developing RTM systems in a way that is independent of the platform specification diversity, making RTM designs portable across different platforms. Our approach uses a formal method to design a high level model of the RTM system, and generate the implementation automatically from the formal model. Formal methods are mathematically-based techniques used for specifying and reasoning about software and hardware systems [6]. We use the Event-B formal method [7] to model and verify RTM systems. The performance parameters and interaction interfaces are instantiated for a specific platform in order to generate a platform-specific RTM implementation from a platform-independent design model. Code generation has been introduced in the Event-B formal method to bridge the gap between abstract specifications and implementation [8]. While the design model is independent of the platforms, the generated code is specific to each platform.

The other challenge associated with the RTM system is its correctness. An RTM mechanism should not compromise the reliability or performance of the platform it is managing. Formal modelling is associated with the verification techniques which can ensure the correctness of the RTM design. The use of formal methods helps to reduce costs by identifying specification and design errors at early development stages when they are cheaper to fix [7].

To validate the portability offered by our approach, we have modelled a platform-independent Reinforcement Learning (RL) based RTM for deadline-based applications and generated platform-specific implementations for three different platforms: ARM Cortex-A8, A7, A15. The impact analysis shows energy saving on the respective platforms. The run-time algorithms are based on the work of [9], which uses prediction for estimating the workload, and RL to select the VF setting; but it [9] does not include any formal approach nor a design model.

In [10] we presented work on automatic code generation of an RTM implementation from a platform-dependent Event-B design model for a specific platform (ARM Cortex-A8) and a video decoder application. In this paper we present a general model-based framework for RTM generation that deals with platform diversity through model parameterisation and customised code generation and we present a more comprehensive validation through experimentation with three different platforms and a wider range of applications. To the best of our knowledge, no work on a formal design of platform-independent RTM systems followed by automatic generation of RTM implementation has been reported.

The paper is structured as follows: Section 2 outlines the platform architectural diversity that motivates our research and an approach to address platform-independent design. Background knowledge including learning-based power management and the Event-B formal method, are presented in Section 3. Section 4 explains our model-based framework for embedded RTM in detail. Finally Section 5 presents the experimental results for three platforms and Section 6 concludes and outlines the future work.

2. Motivation: Addressing Platform Specification Diversity

Our RTM approach uses Reinforcement Learning (RL) [11] to achieve optimal decisions for VF settings. An RL based RTM is highly dependent on hardware platform specifications. The objective of RL is to learn and make better decisions under workload variation. In the exploration stage, random actions are taken and the corresponding responses (rewards or penalties) are recorded in a lookup table called a *QTable*. In the exploitation stage, the decisions that can achieve highest rewards are applied. Decisions in RL terminology are known as *actions* and the workloads are known as *states*. This information is stored as rows and columns in the QTable.

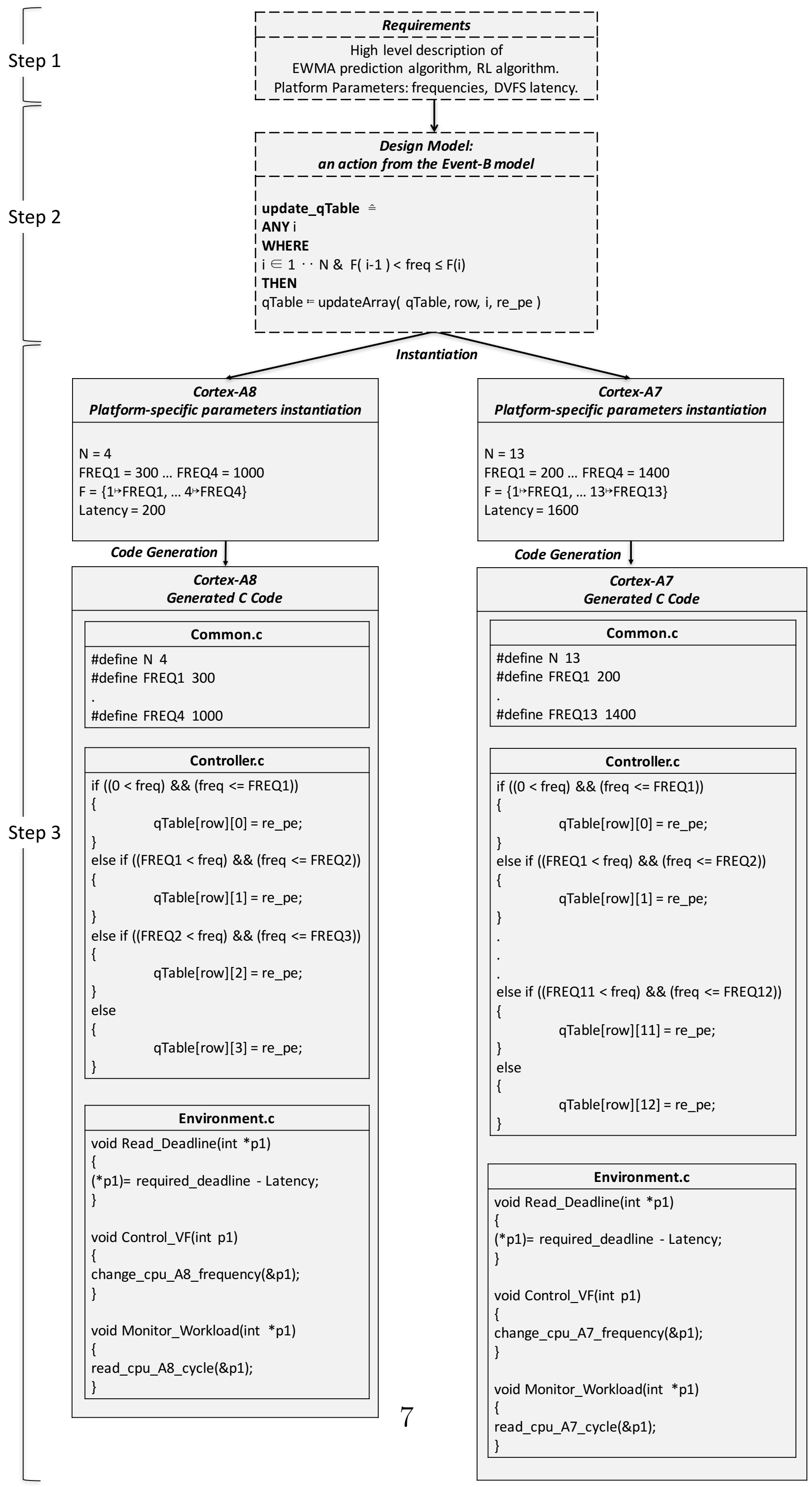
To implement the RL correctly in various platforms, the differences between the platforms need to be identified. We have implemented a video decoding application in three different platforms: ARM Cortex-A8, A7 and A15 processors. Table 1 shows the platform specific parameters including number of VF pairs, DVFS switching latency and relative performance of each platform measured by Cycles Per Instruction (CPI) normalised to that of A8. CPI can be different for different applications (due to different instructions), and the CPI data presented in Table 1 represents average CPI measurements based on the video decoder application. These platform specific parameters can influence the implementation of the RL algorithm, for example the size of the QTable is different for each platform because of the difference in the number of VF pairs (2nd column) and the CPI (last column). In addition, the switching between different VF pairs is not instant, and the DVFS switching latency (3rd column) needs to be subtracted from the deadline when calculating rewards and penalties.

Table 1: Platform-specific parameters

Platform	**Num of VF pairs**	**DVFS latency (milli second)**	**Performance (Relative CPI)**
Cortex-A8	4	0.2 ms	1
Cortex-A7	13	1.6 ms	1.37
Cortex-A15	19	1.1 ms	0.81

The other difference is to do with the interfacing functions between the application, runtime manager and hardware such as reading the deadline, controlling VF settings and monitoring the workload. All these factors will affect the implementation of the RTM code. To ensure the correct functionality of RTM code, a systematic approach is needed to identify the difference in platform parameters and generate the correct implementation for each platform.

To address platform-independent RTM development, we propose a framework in which the RTM design model is independent of the platform diversity, and the RTM implementation can be automatically generated specifically for each platform. Figure 1 shows an example of the framework being used for two different platforms: a Cortex-A8 and A7. The generic framework, and details of the steps, will be explained later in Section 4. From top to bottom, Figure 1 illustrates 1) requirements, 2) part of the Event-B design model concerning RL QTable update, and 3) the corresponding generated C code including the specific interaction interfaces, for Cortex-A8 and A7 platforms separately. We start from the high level description of the RTM algorithms and platform parameters. In Step 1, the RTM requirements are identified and documented. In Step 2, an Event-B design model is constructed from the requirements. In Step 3, the design model is instantiated for the Cortex-A8



High level description of
EWMA prediction algorithm, RL algorithm.
Platform Parameters: frequencies, DVFS latency.

```
update_qTable ≙
ANY i
WHERE
i ∈ 1 ‥ N & F( i-1 ) < freq ≤ F(i)
THEN
qTable ≔ updateArray( qTable, row, i, re_pe )
```

Cortex-A8 — Platform-specific parameters instantiation

```
N = 4
FREQ1 = 300 … FREQ4 = 1000
F = {1↦FREQ1, … 4↦FREQ4}
Latency = 200
```

Cortex-A7 — Platform-specific parameters instantiation

```
N = 13
FREQ1 = 200 … FREQ4 = 1400
F = {1↦FREQ1, … 13↦FREQ13}
Latency = 1600
```

Cortex-A8 — Generated C Code

Common.c

```
#define N 4
#define FREQ1 300
.
#define FREQ4 1000
```

Controller.c

```
if ((0 < freq) && (freq <= FREQ1))
{
        qTable[row][0] = re_pe;
}
else if ((FREQ1 < freq) && (freq <= FREQ2))
{
        qTable[row][1] = re_pe;
}
else if ((FREQ2 < freq) && (freq <= FREQ3))
{
        qTable[row][2] = re_pe;
}
else
{
        qTable[row][3] = re_pe;
}
```

Environment.c

```
void Read_Deadline(int *p1)
{
(*p1)= required_deadline - Latency;
}

void Control_VF(int p1)
{
change_cpu_A8_frequency(&p1);
}

void Monitor_Workload(int *p1)
{
read_cpu_A8_cycle(&p1);
}
```

Cortex-A7 — Generated C Code

Common.c

```
#define N 13
#define FREQ1 200
.
#define FREQ13 1400
```

Controller.c

```
if ((0 < freq) && (freq <= FREQ1))
{
        qTable[row][0] = re_pe;
}
else if ((FREQ1 < freq) && (freq <= FREQ2))
{
        qTable[row][1] = re_pe;
}
.
.
.
else if ((FREQ11 < freq) && (freq <= FREQ12))
{
        qTable[row][11] = re_pe;
}
else
{
        qTable[row][12] = re_pe;
}
```

Environment.c

```
void Read_Deadline(int *p1)
{
(*p1)= required_deadline - Latency;
}

void Control_VF(int p1)
{
change_cpu_A7_frequency(&p1);
}

void Monitor_Workload(int *p1)
{
read_cpu_A7_cycle(&p1);
}
```

Figure 1: Addressing platform specification diversity: Cortex_ARM8 and ARM7

platform and for the Cortex-A7 platform and corresponding C implementations are generated.

Figure 1, Step 2 presents one of the events of the design model, *update_qTable*, which updates the qTable array. The event specifies that a value i (indicated by the keyword **ANY**) should be selected that satisfies the event guards (indicated by the keyword **WHERE**) and the update action (indicated by the keyword **THEN**) should be performed with the selected value for i. Here i represents an index for a frequency value and determines which column of the qTable gets updated with the value of the reward/penalty (re_pe).

Figure 1, Step 3 the independent model is instantiated by defining concrete values for the model parameters for the Cortex-A8 platform and the Cortex-A7 platform. For the Cortex-A8 on the left, the number of VF pairs is 4, whereas for the Cortex-A7 on the right, it is 13. The values of frequencies, an ordering function, called f, and the DVFS latency value are also instantiated.

The code generation result contains three files: *Common.c* contains the data definition, *Controller.c* contains the C code of the RTM algorithms, and *Environment.c* contains the interaction interfaces. As explained earlier in this section, the number of frequencies corresponds to the number of *qTable* columns. The Event-B model is independent of these platform parameters but *Controller.c* is dependent on the number of frequencies. The event is translated to a set of "if_then_else" branches in the number of frequencies (number of *qTable* columns), to modify the *qTable*. The left *Controller.c* box presents the generated implementation for the Cortex-A8 by 4 branches modifying the 4 columns of the *qTable* depending on the value of selected frequency. For the Cortex-A7 (left box), there are 13 branches to modify the columns of *QTable* by 13 columns. The variable *row*, specifying the appropriate row of the *qTable*, has been assigned in a separated event according to the value of the predicted workload. Each *Environment.c* includes platform-specific API calls in the interaction interfaces to the application layer and the hardware layer. For example, *change_cpu_A8_frequency*, a predefined API for the Cortex-A8, is called in the body of generated *Control_VF* interaction interface in order to change the frequency of the Cortex-A8 hardware.

Both implementations are *automatically* generated from the same model, even though one has 4 branches and the other has 13. In contrast to the automatic generation, modifying one version of an implementation to a different number of branches *manually*, would require re-coding and can be error-prone.

The presented model-based framework to build the RTM is intended to achieve increased productivity of RTM software in embedded systems. Our previous work [10] presents our initial effort to apply formal methods in embedded software area and the outcome model was specific to one platform, whereas the proposed framework in this paper is demonstrating a general platform-independent model. The Platform Independent (PI) design model is reusable across different platforms with diverse core characteristics. Platform Specific (PS) core characteristics are used to instantiate the PI design model to be transformed to the PS executable software. Moreover the framework addresses the correctness of the RTM design; the Event-B formal model is verified using theorem proving and model checking to ensure the correctness of the modelled properties and consistency between different refinement levels of the design model.

3. Background

3.1. Learning-based RTM

In this paper, we apply our approach to an RTM that manages applications with epochs of varying workload, e.g., each frame in a video decoder is an epoch and workload varies between frames. Our RTM algorithm [9] 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 VF setting so that the predicted workload can finish execution before the epoch deadline. After the epoch has completed, the RTM learns by using feedback to update 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) [12]. For the Decision Making, Reinforcement Learning (RL) is used [11], using the Q-Learning algorithm. The objective of RL is to learn how to make better decisions under variations. Decisions in RL terminology are known as actions, and the environment is represented as states.

The RTM algorithm is shown in Algorithm 1. For every new epoch, the RTM first predicts the workload, based on this it selects a VF value. After processing the frame, the performance is determined to fine tune the prediction and the decision algorithms.

Exploration and Exploitation Phases: initially there is no knowledge of the system workloads, so the decision algorithm must start exploring deci-

Algorithm 1 RTM Algorithm

```
for every New Epoch do
    Prediction_Unit.PredictWorkload
    Decision_Unit.MapWorkload
    Decision_Unit.SelectPowerState(VF)
    Prediction_Unit.UpdatePrediction
    Decision_Unit.UpdateQ-Table
end for
```

sions 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 penalised. Actions in this context, are the VF pairs, and states are the different amounts of workload the system may have. It is important to note that the VF pairs are discrete, so the *best* decision may not be optimal, but it is the best among the VF 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 (Cortex-A8), the *best* decision is to execute the workload at 600MHz. The 'best' in the context of this paper is defined as the lowest VF pair that fulfils the performance requirement. Initially, the decisions of the algorithm are not optimal. However, after several epochs[1] the accuracy in the selected action improves and the algorithm always selects the best action in a given state.

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

This phase of the algorithm is called the *Exploitation phase*. The learning algorithm [11] also penalises in case of system overload, even at the highest frequency. However the penalty is proportional to the deadline miss time, therefore even though running at the highest frequency incurs a penalty, it will be smaller than running at lower frequencies. In this paper, we have not modelled the system overload and so we do not support penalisation for it.

3.2. Event-B Formal Method

Event-B [7] is a formal method for system-level modelling and analysis which allows us to produce a precise formal model of the RTM algorithms that abstract away from platform dependent parameters and interfaces. 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. Instead of building a single big model which can be complex and error-prone, Event-B refinement allows us to build the model gradually by introducing details of the system in each refinement level. Therefore we can verify the correctness of a model step by step. The Rodin platform [13] is an Eclipse-based IDE for Event-B that provides support for modelling and mathematical proof.

A model in Event-B can consist of several *Contexts* and *Machines*. Contexts contain the static part (types and constants) of a model while machines contain the dynamic part (variables and events). Contexts provide axiomatic properties of an Event-B model, whereas Machines provide behavioural properties. A Machine consists of variables, invariants and events. An invariant is a predicate or constraint, which every state in the model must satisfy. Each event is composed of a name, a set of guards G(t,v) and some actions S(t,v), where t are parameters of the event and v is state of the system which is defined by variables. All events are atomic and can be executed only when their guards hold.

The correctness of an Event-B model is defined by invariant properties. 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 available in the Rodin toolset.

Theorem Proving: There are different POs which are generated by Rodin during development of a system [14]. The most important 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 verifying 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 [15] is an animator and model checker for Event-B. ProB allows fully automatic exploration of Event-B models restricted to finite states., and can be used to systematically check a specification for a range of errors.

250 4. The Model-Based Framework

Figure 2 illustrates our layered model-driven framework in which our RTM design, providing generic (platform-independent) functionality across different platforms, can be selectively instantiated by additional user defined platform specific parameters and interfaces. The dashed lines indicate the Platform Independent (PI) components, while the solid lines present the Platform Specific (PS) components.

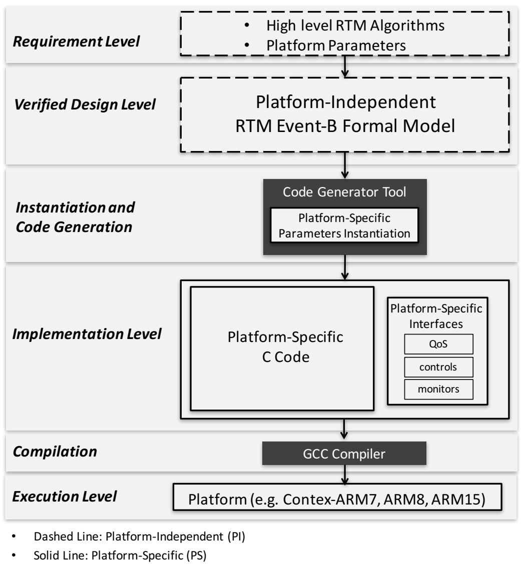

Requirement Level
• High level RTM Algorithms
• Platform Parameters
Verified Design Level
Platform-Independent RTM Event-B Formal Model
Instantiation and Code Generation
Code Generator Tool
Platform-Specific Parameters Instantiation
Implementation Level
Platform-Specific C Code
Platform-Specific Interfaces
QoS
controls
monitors
Compilation
GCC Compiler
Execution Level
Platform (e.g. Contex-ARM7, ARM8, ARM15)
• Dashed Line: Platform-Independent (PI)
• Solid Line: Platform-Specific (PS)

Figure 2: A framework to automate code generation of a platform independent RTM from a high level formal model

Following is a summary of the layers:

The requirement level: Starting from the top layer, the requirement level includes the PI high level description of RTM algorithms, such as a high level description of the RL in Figure 1, and platform parameters, such as the number of frequencies in Figure 1. High level requirements provide knowledge to make the formal design model in the next step.

The verified design level: Our design methodology uses the Event-B formal method to create a verified PI model of the RTM system using incremental refinement.

Code generation: The gap between the design level and implementation level is bridged by the code generation tool which automatically transforms the instantiated design model into the executable C code. The code generation in performed by the code generation plugin [8] in the Rodin platform.

The implementation level: The generated RTM implementation is specific to each platform as well as the interfaces to access the QoS, control knobs such as VF setting and monitor core activities.

The execution level: Finally the RTM implementation is complied by GCC compiler and executed in the platforms using Hardware Abstraction Layers (HAL) in the Linux operating system.

We have applied our framework to develop a Reinforcement Learning (RL) RTM system for applications with soft deadlines. The Event-B design model of RL RTM system is instantiated per HW platform and the RL algorithm is automatically generated from the instantiated Event-B design model of the RTM system for each platform. This section presents details of each framework level separately.

4.1. Requirement Level

According to the framework illustrated in Figure 2, our **requirement level** includes the high level descriptions of the platform-independent RTM algorithms and platform parameters. We outline requirements on the RTM algorithms in this section and the corresponding design model is outlined in the next section.

An overview of the RTM is illustrated in Figure 3. First the application provides the required deadline, e.g., frames-per-second (FPS) for video decoding, to the RTM; then the optimal value of VF is decided by the RTM. The RTM controls the VF in the hardware and the frame is executed in the hardware. After that the actual value of workload to decode the frame is monitored.

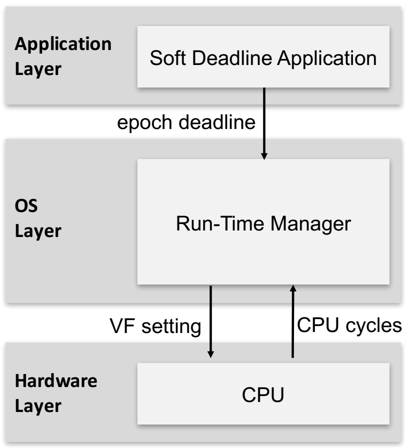


Figure 3: Run-Time Management system: a cross-layer approach

To achieve the required deadline set by the application, we use the learning approach outlined in Section 3.1. Details of the prediction and learning algorithms are explained and modelled next.

4.2. Design Level

The top level of Figure 4 illustrates our design architecture for Event-B modelling of the RTM. This figure presents details of the **verified design**

level in Figure 2.

As shown in Figure 4, the Event-B model of the RTM system comprises an abstraction level and two refinement levels. In the abstract model we focus on the main functionalities of the RTM system including the variables and actions modelling the interaction of the RTM with the application in a platform-independent way. The abstract model is followed by two levels of refinement, where the workload prediction and the RL 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 interaction interfaces between the RTM and the application and hardware. By separating controller behaviour and environment behaviour, the representation of the RTM and, the application/hardware are divided. This structure is used for code generation configuration, where the controller translation consists of RTM algorithms, and the environment translation represents the interfaces to the application and hardware. Details of this implementation are explained in Section 4.3. The sub-models need some preparation before the final step of being translated to the executable code. These preparations are included in refinement of sub-models: Controller tasking and Environment tasking. In these refinements, the sequencing and branching of Event-B actions are defined. Also additional translation rules are defined to translate the Event-B mathematical operators to the corresponding C operators.

Our RTM design model is independent of the platform (indicated by

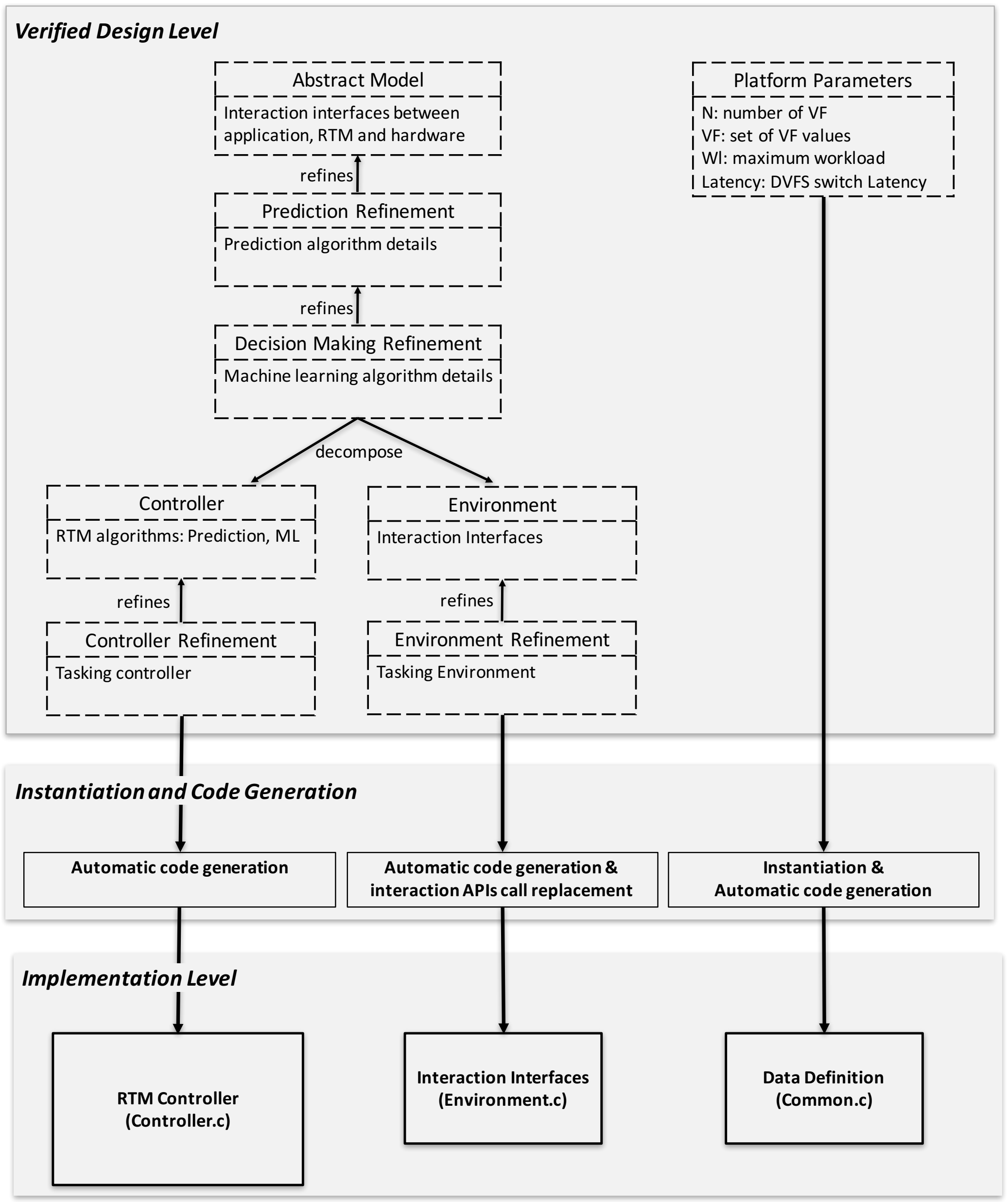

Verified Design Level
Abstract Model
Interaction interfaces between application, RTM and hardware
refines
Prediction Refinement
Prediction algorithm details
refines
Decision Making Refinement
Machine learning algorithm details
decompose
Controller
RTM algorithms: Prediction, ML
Environment
Interaction Interfaces
refines
refines
Controller Refinement
Tasking controller
Environment Refinement
Tasking Environment
Platform Parameters
N: number of VF
VF: set of VF values
Wl: maximum workload
Latency: DVFS switch Latency
Instantiation and Code Generation
Automatic code generation
Automatic code generation & interaction APIs call replacement
Instantiation & Automatic code generation
Implementation Level
RTM Controller (Controller.c)
Interaction Interfaces (Environment.c)
Data Definition (Common.c)

Figure 4: Design Level and Implementation Level

dashed lines in Figure 4). The platform parameters, including number of supported VF, maximum workload and DVFS switch latency, are used to instantiate the model to generated platform-specific C code for each platform. The C code is generated automatically from the final refinement models of the controller and the environment. The next section will explain the implementation level (bottom level of Figure 4). All of design steps, including the abstraction, refinements, model decomposition and code generation are verified using the Rodin toolset.

4.2.1. Abstraction

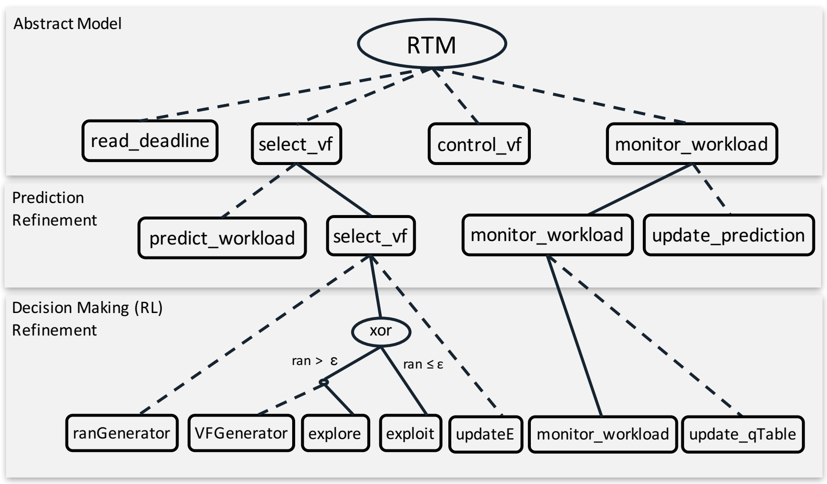


Figure 5: Design Level Details: Event Refinement Structure for the RL-based RTM Event-B model

To visualise the events of the Event-B abstract/refinement levels, presented in the top level of Figure 4, we use an approach, called Event Refinement Structures (ERS) [16]. The ERS of the RTM Event-B model is presented in Figure 5 which is divided into three regions indicating the refinement levels: the abstract model, prediction refinement and decision making refinement. The top region shows the abstract level including four events. Each node indicates an event in the Event-B model and the oval contains a name for the overall flow of events in the model. The nodes are read from left to right indicating the ordering between them. First the *read_deadline* event executes followed by execution of *select_vf*, *control_vf* and *monitor_workload*.

Abstractly the value of the VF 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* $\widehat{=}$

`act1` : $freq :\in VF$

The set *VF* is instantiated specifically for each platform. For example, for the ARM Cortex-A8, which provides four values of *VF*, it is defined as follows:

VF := { FREQ1, FREQ2, FREQ3, FREQ4 }

4.2.2. Prediction Refinements

In the abstract level, we do not model details of the workload prediction nor the decision making. In a refinement level (the middle region of Figure 5), the details of the prediction algorithm are added to the abstract events: *select_vf* and *monitor_workload*.

The *select_vf* event is refined into two concrete events: *predict_workload*, where the workload is predicted and *select_vf*, where the value of VF is decided based on our prediction. The *monitor_workload* event is also refined into two events: *monitor_workload* (monitoring the actual workload) and *update_prediction* (updating the prediction factors). In ERS, the line types indicate whether the corresponding event is a refining event (solid line) or a new event (dashed line). In refining the *select_vf* event, *predict_workload* is a new event and the concrete *select_vf* event refines the abstract *select_vf*.

The prediction algorithm estimates the workload for the next frame using a modified form of Exponential Weighted Moving Average (EWMA). The EWMA algorithm is widely used in the literature [17, 1, 18] because of its lightweight implementation.

The EWMA predictor is modelled in two levels of refinements. In the first level, the predictor is defined in terms of the full history of measured workloads; and in the second level, the predictor generates a prediction of the future value based on the average of the previous values weighted exponentially, where the most recent values have greater weight than the older ones. In Section 4.2.4, it is proved that the second definition is a correct refinement of the first one.

In the first refinement of the prediction, the specification of *predict_workload* and *update_prediction* events are as follows:

Event *predict_workload* $\widehat{=}$

act1 : $pwl := predict(l, n, wl\_hst)$

Event *update_prediction* $\widehat{=}$

act1 : $wl\_hst := wl\_hst \cup \{n \mapsto w\}$

act2 : $n := n + 1$

l is a constant specifying the weighting factor, n and ws_hst ($ws\_hst : (1..n) \rightarrow INT$) are variables specifying a frame counter and history of measured workloads respectively. In the *predict_workload* event, the *predict workload* variable (*pwl*) is assigned to the predicted value through the *predict* operator from the EWMA theory. In the *update_prediction* event, the history of the workloads (*wl_hst* variable) is updated to include the last (n^{th}) monitored actual workload (w variable).

A *theory* is an Event-B component where we can introduce new mathematical operators. In this development, we have defined a theory of EWMA where the prediction operators are defined. The *predict* operator is defined in terms of the full history of measured workloads, with three arguments as follows[2] ($\mathbb{Z}$ is the set of natural numbers):

Theory EWMA
operator $predict\ (l \in \mathbb{Z},\ index \in \mathbb{Z},\ w \in \mathbb{Z} \nrightarrow \mathbb{Z})\ \widehat{=}$

$$l * SUM(\lambda i.i \in 0..index-1 \mid w(i) * (i-l)\ exp\ (index - i))$$

Here *w(i)* is the actual workload (for the i^{th} frame).

In the second refinement of the prediction, the specification of *predict_workload* and *update_prediction* events are as follows:

Event *predict_workload* $\widehat{=}$
refines *predict_workload*

act1 : $pwl := avgwl$

Event *update_prediction* $\widehat{=}$
refines *update_prediction*

[2]A conventional representation of the abstract predict operator is:

$$l \cdot \sum_{i=0}^{n-1} w(i) \cdot (i-l)^{n-i} \text{ where } 0 \le l \le 1 \qquad (1)$$

`act1` : $avgwl := update(l, w, avgwl)$

In the *predict_workload* event, the *pwl* variable is assigned to the *average workload* variable (*avgwl*); where *avgwl* is updated in the *update_prediction* event, according to the definition of *update* operator in the EWMA theory:

Theory EWMA
operator $update(l \in \mathbb{Z},\ w \in \mathbb{Z},\ avgwl \in \mathbb{Z})\ \widehat{=}\ l * w + (1 - l) * avgwl$

Using the Event-B proof techniques (Section 4.2.4), we verify that the abstract definition, based on the full history of actual workloads, is correctly refined by maintaining a running average. The abstract definition is more clear and thus easier to validate. The refined definition is much more efficient to implement.

The value of *freq* is calculated based on the predicted workload in *select_vf* event:

Event *select_vf* $\widehat{=}$
refines *select_vf*

`act1` : $freq := pwl * fps$

This event is refined in the next refinement (decision making) where the *freq* is selected based on the decision making algorithm.

4.2.3. Decision Making (Reinforcement Learning) Refinement

The bottom region of Figure 5 shows a further refinement, where details of RL are modelled. The *select_vf* event and *monitor_workload* event are refined to include the details of the RL.

At the bottom region of Figure 5, the *select_vf* event is refined to specify the exploration and exploitation phases. Reading the children of the *select_vf* node from left to right, first the *ranGenerator* event nondeterministically generates a value (implemented as a random choice). Comparing this nondeterministically chosen value with the exploration-exploitation ratio (ϵ), either the *explore* or *exploit* event are executed and this is followed by updating ϵ (*updateE* event). The oval containing *xor* represents an exclusive choice between its branches. In case of exploration, first the *VFGenerator* event nondeterministically choses a VF value within the available VFs to be used in the *explore* event. 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[3]. The availability of ϵ makes 're-learning' a feasible operation, especially for dynamic systems in which the best action for a particular state may change gradually. If relearning is needed, the ϵ may be reduced to allow for more exploration to take place.

Below is the Event-B description of the *explore* and *exploit* events. These events are guarded based on the value of the *random* variable (nondeterministically chosen 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 VF value (generated in the *VFGenerator*). The *exploit* event assigns *freq* value into the optimal value of VF according to the predicted workload (*pwl*). *optimalVF* is an operator defined in a theory where all of the necessary RL

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

operators are defined.

Event *explore* $\widehat{=}$
refines *select_vf*

when
`grd` : $random > epsilon$
then
`act1` : $freq := randomVF$

Event *exploit* $\widehat{=}$
refines *select_vf*

when
`grd` : $random \leq epsilon$
then
`act1` : $freq := optimalVF(QTable, pwl)$

Updating Phase: knowledge generated from learning is stored as values in a QTable, which is a lookup table with values corresponding to all State-Action pairs. At each decision epoch, the decision taken for the last epoch is evaluated; the reward or penalty computed is added to the corresponding QTable entry, thereby gaining experience on the decision. This reward/penalty is calculated with a cost function, which in this RTM context is defined as:

$$re\_pe = \begin{cases} \frac{100t}{d} & \text{if } t \leq d \\ -\frac{100(t-d)}{3d} & \text{if } t > d \end{cases} \tag{2}$$

where *re_pe* is the reward/penalty, t is the runtime and d is the deadline. At the bottom region of Figure 5, the *monitor_workload* event is refined to include the *update_qTable* event, where the workload is rewarded or penalised. The Event-B specification of the *update_qTable* event is as follows:

Event *update_qTable* $\widehat{=}$

any

i

when

`grd` : $i \in 1..N \land F(i-1) < freq \leq F(i)$

then

`act1` : $qTable := updateArray(qTable, row, i, re\_pe)$

In the body of the event, the value of the variable *qTable* is updated, where *qTable* is defined as a two-dimension array, *row* specifies the row number, i specifies the column number and *re_pe* is the reward or penalty of the most recent decision. The value of the *row* and *re_pe* are assigned in the separated events. The guard is defined as can be seen for the code generation purposes (details described in Section 4.3).

The definition of *updateArray* is specified as an operator in the *Array* theory including the corresponding C translation as follows:

Translator

Target C

MetaVariables

: $a \in \mathbb{Z}$

: $r \in \mathbb{Z}$

: $c \in \mathbb{Z}$

: $v \in \mathbb{Z}$

`Formula` : $a := updateArray(a, r, c, v)$

`Output:` : $a[r][c] = v$

The *Ouptput* specifies the translation of the Event-B *Formula* to the appropriate syntax in the C programming language.

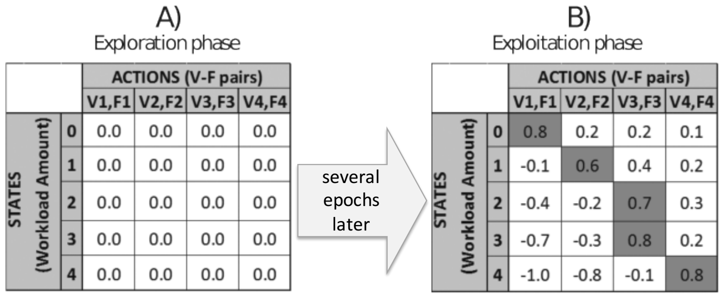
The value of the *re_pe* variable is assigned in *cost_reward_assign* and *cost_penalty_assign* events. The *cost_reward_assign* event is as follows:

Event $cost\_reward\_assign \mathrel{\widehat{=}}$
when
 grd : $(w/freq) \leq d$
then
 act1 : $re\_pe := min(1, cost\_reward$
 (w, freq, d))

The *cost_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, meaning the deadline is achieved and the *QTable* needs to be rewarded. The *cost_reward* is defined as an operator in the RL theory based on Equation 2
.

Figure 6 shows the evolution of the QTable. Initially, the values in the QTable are all zeros (Figure 6(A)). In the exploration phase the QTable will be filled with values indicating rewards or penalties (Equation 2)). In the exploitation phase, the 'best' actions are determined based on the QTable entries with highest rewards (highlighted in Figure 6(B)).

Model Decomposition: as shown in Figure 4, the final refinement is divided into two smaller sub-models. The controller sub-model includes the RTM actions: *predict_workload*, *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_workload*.


A)
Exploration phase
several epochs later
B)
Exploitation phase

STATES (Workload Amount)	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

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

Figure 6: QTable during A) exploration and B) exploitation phases. The highlighted boxes represent the best Action for each State.

4.2.4. Verification

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, mostly associated with correct sequencing of events. A manually proved PO is presented here as an example of verification.

As presented in Section 4.2.2, the prediction refinement consists of two levels. The following invariant captures the relationship between the *avgwl* variable of the refinement and the workload history (*wl_hst*) of the abstract model:

inv1: $avgwl = predict(l, n, wl\_hst)$

This invariant is required to prove that the action of the refined prediction event correctly implements the abstract event. To prove this invariant, we introduce the following theorem which shows the algebraic connection between abstract *update* operator and the concrete *predict* operator:

thm1: $\forall n, w \cdot n > 0 \land w \in \mathbb{Z} \Rightarrow$

$update(l, w, predict(l, n, wl\_hst)) = predict(l, n + 1, wl\_hst \cup \{n \mapsto w\})$

The invariant and theorem are proved interactively with the Rodin theorem prover.

We also analysed our model using ProB to ensure that the model is deadlock free and convergent. At any point during model checking, at least one of the events of the model should be enabled to ensure that the model is deadlock free. For each new event added in the refinements, we have verified that it would not take control forever (convergence). Also *INV* POs ensure that the new events keep the existing ordering constrains between the abstract events. The ordering between events are specified as invariants, the PO associated with each invariant ensures that its condition is preserved by each event.

4.3. Code Generation and Implementation Level

The Event-B model of the RTM system is automatically translated to executable C code using the code generation plugin of the Rodin toolset. The bottom level of Figure 4 illustrates the procedure of generation of RTM software to be executed on the hardware. To generate code, the controller is instantiated by the platform-specific parameters for one platform and translated to the "Controller.c" file. The platform specific parameters are translated to the C variable definitions in the "Common.c" file. The environment, modelling the interactions, is translated to the signature of C functions, representing the interactions. Since in the independent design model we abstracted from details of interactions between the RTM and application and

HW layers, the specific interaction APIs for each platform needs to be called in the generated environment file. Our experimental results from executing generated code in various hardware platforms, are presented in Section 5.

How Code Generation works: As shown in Figure 4, 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_workload;
update_avgwl;
if cost_reward_assign
else cost_penalty_assign;
update_qTable;
```

This indicates the ordering between events *monitor_workload* and *update_avgwl* followed by a branching between *cost_reward_assign* event and *cost_penalty_assign* event. Below is part of the result of automatic code generation for the ARM_A8 platform with 4 values of frequencies ($N = 4$ in the *update_qTable* event):

```
Env_monitor_actwl(&actwl);
avgwl = (l * w + (1 - l) * avgwl);
if (w / freq <= d) {
   re_pe = min(1, (100 * w) / (freq * d));
} else {
   re_pe = max(-1, -((w / freq - d) * 100) / (3* d));
}
if ((0 < freq) && (freq <= FREQ1)) {
          qTable[row][0] = re_pe;
} else if ((FREQ1 < freq) && (freq <= FREQ2)) {
        qTable[row][1] = re_pe;
} else if ((FREQ2 < freq) && (freq <= FREQ3)) {
        qTable[row][2] = re_pe;
} else {
        qTable[row][3] = re_pe;
}
```

First the *monitor_workload* is translated to a call to the environment function. Then *update_avgwl* is translated into the second line according to the operator definition for the *update*. Finally branching is generated on the *cost_reward_assign* and *cost_penalty_assign* depending on the event guards. The Event-B guard of the event is translated into the branching condition in C. *re_pe* is assigned according to the definition of cost function operators in the RL theory (according to the Equation 2). And the rest of the code is the translation of the *update_qTable* event which has been explained earlier in Figure 1. The choice of i (an index representing the choice of frequency) is translated to a set of "if_then_else" branches in the number of frequencies (number of *qTable* columns), to modify the *qTable*.

As shown in Figure 1, the generated environment is similar for both platforms. It includes the function signatures for the interfaces to read the deadline (*Read_Deadline*), control VF (*Control_VF)* and monitor the workload (*Monitor_Workload*). In a further step, the bodies of these interface functions are replaced by calling the right platform specific APIs. For example the variable *required_deadline* needs to be subtracted from the platform dependent DVFS switching delay *Latency* which is 200 *us* for Cortex-A8 and 1600 *us* for Cortex-A7 (Table 1). Functions *change_cpu_A8_frequency* and *change_cpu_A7_frequency* implement the hardware frequency changes for

Cortex-A8 and A7 respectively to accommodate the difference in number of VF pairs. It first checks if the required frequency is in the VF table for the associated platform and make the change accordingly. *read_cpu_A8_cycles* and *read_cpu_A7_cycles* use platform specific assembly instructions to read the cycle counter for Cortex-A8 and Cortex-A7 respectively. These API functions need to be implemented specifically for each platform to address the differences in both OS and hardware controls. The next section will describe the adopted architecture for the generated RTM at the OS layer.

4.4. Execution Level and Hardware Abstraction Architecture

As discussed in previous sections, 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). Dividing into *Controller* and *Environment* sub-models is in-line with the HAL[4] (Hardware Abstraction Layers) principle [19]; The *Controller* sub-model is abstracted from platform dependencies, while the *Environment* sub-model provides platform-specific calls to get/set monitors/knobs. An interface to provide these functions has been designed. Figure 7 shows the modified RTM diagram from Figure 3, where the box in the centre represents the generic RTM auto generated code, and the highlighted boxes provide the interactions with

[4]In computers, a hardware abstraction layer (HAL) is a layer of programming that allows a computer operating system to interact with a hardware device at a general or abstract level rather than at a detailed hardware level

the hardware and application layers. The translated environmental functions call these interaction interfaces.

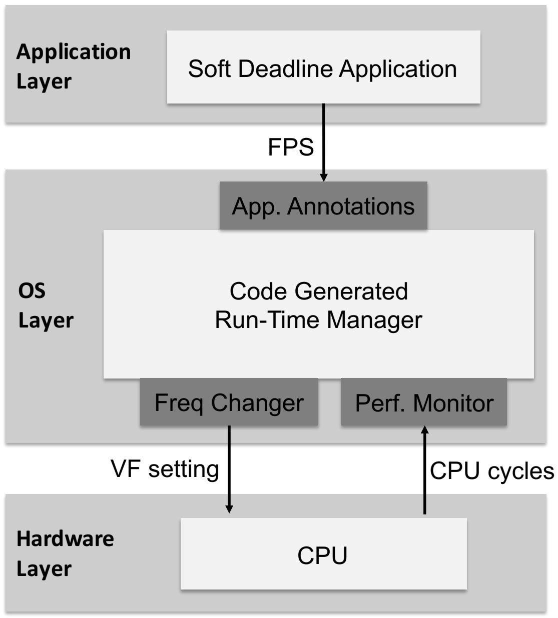


Figure 7: Code generated RTM architecture for a deadline-based application

Hardware Abstraction Architecture: In order for the generated RTM to sit at the OS layer, it has been implemented as a Linux Governor [20] through a Loadable Kernel Module (LKM), which provides the interface and drivers to make the VF changes and monitoring workload. This Governor provides the three interfaces needed for the algorithm: the *Frequency Changer*, the *Performance Monitor* and the *Application Annotations*.

As part of the HAL in Linux, the RTM implementation uses *sysfs* for both the performance counter module and the RTM module (*sysfs* makes device information available as virtual files). This allows the RTM module to interact with the performance counter module, configuring it and requesting for performance data. As the RTM module and performance counter module are implemented as *sysfs*, they are visible at user space so the user can read/write parameters and data to the modules.

The Frequency Changer provides the *CPUFreq* drivers to change the VF 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 ARMv7 (ARM Cortex-A8, Cortex-A7 and Cortex-A15), which provides Performance Monitoring registers [21]. ARMv7 assembly code was used to access these monitors in the LKM. The Application Annotations interface provides a library for the application to send its performance requirement (FPS) to the RTM using function *config_governor(int fps)*. It also provides function calls to trigger the Governor to start (*start_governor()*) and to finish working (*stop_governor()*). It notifies the RTM of a new frame start through function *new_epoch()*. This communication is done through *ioctl* calls (device specific input/output calls). After the RTM C code is generated, it is cross-compiled with the installed Linux and processor architecture to create the respective LKM. When the LKM is loaded, it waits for the *read_deadline* from the application and the *start_governor* calls to start working. The *new_epoch* call at every new frame triggers the RTM algorithm both for deciding the new VF and learning (from the previous frame): the deadline is compared with the actual runtime to update the learning table with the reward/penalty (Equation 2). At the end of the application, the *stop_governor* call ends the RTM execution.

The deadline $t_{deadline}$ is given by the frame rate fps, so:

$$t_{deadline} = \frac{1}{fps} \tag{3}$$

The time taken to process the frame (t_{frame}) is obtained by getting $timestamp(n)$ of the global system clock given in microseconds every frame, obtaining the difference with the previous frame $timestamp(n-1)$. The t_{frame} is then compared with $t_{deadline}$ to decide whether the deadline was passed or not:

$$t_{frame} = timestamp(n) - timestamp(n-1) \tag{4}$$

$$deadline\_passed = \left\{ \begin{matrix} 1 & \text{if } t_{frame} \leq t_{deadline} \\ 0 & \text{if } t_{frame} > t_{deadline} \end{matrix} \right\} \tag{5}$$

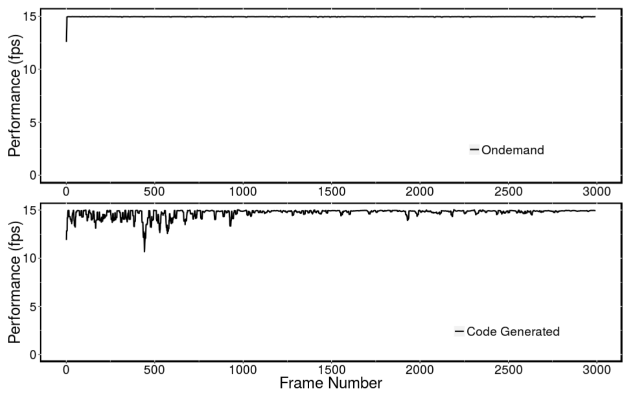
5. Experimental Results and Evaluation

Our experiments demonstrate that we can automatically generate different platform-specific software for different architectures, from the same platform-independent model and observe the effectiveness of the generated implementations in terms of energy management. We validate our work experimentally for three different platforms and two applications in terms of performance and power consumption. Experiments were conducted on the BeagleBoard-xM with Cortex-A8 processor and ODROID-XU3 with both Cortex-A7 and Cortex-A15 processors. Both platforms were running the Linux operating system. Our RL based RTM targets applications with soft deadlines including multi media and computer vision applications. For our experiment, the test applications are a video decoder and a Jacobian matrix solver.

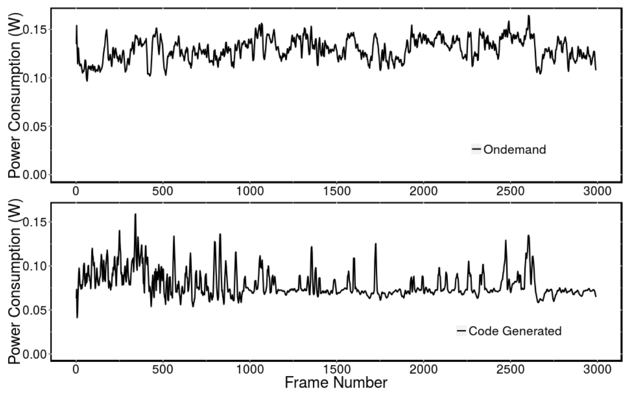
The video decoding is based on FFmpeg libraries using the H.264 codec at a VGA resolution (640x480). On the Cortex-A8 it was running for 720 frames while on the Cortex-A7 and Cortex-A15 it was 3000 frames. More frames are needed due to a longer learning period on the Cortex-A7 and Cortex-A15 due to a larger QTable caused by more VF pairs. The video decoding application is annotated to communicate with the Governor through *ioctl* calls such as `config_governor(int fps)`, `start_governor()`, `stop_governor()` and `new_epoch()` as shown in Section 4.4.

We also used the RTM for an application with different characteristics to the video decoder on an ODROID-XU3 Cortext-A15 platform: a Jacobian matrix solver followed by least-squares solution computation, targeting 10 solutions per second with a 1024x1024 randomly seeded matrix. This application demonstrates higher compute load but lower frame rate than the video decoder.

Figure 8 shows the comparison of performance and power consumption for the Cortex-A7 (video decoding application) when using our code generated RTM against the Linux-ondemand governor. Figure 8.(a) shows the effectiveness of the generated RTM with a performance constraint of 15 FPS. The ondemand governor occasionally misses the deadline, while the generated RTM perform worse in the beginning because of learning) but gradually improved to achieve almost the same performance as the ondemand governor. Figure 8.(b) shows the power consumption in turn for each frame. This shows the different VF controls during two phases (exploration and exploitation) of the RTM. It can be seen during the exploration phase, low VF settings tend to cause performance losses. During the exploitation phase the generated RTM achieves similar performance with significant power savings when compared to the ondemand governor. This behaviour is comparable to the one presented in [9] for hand written RTM code, where exploration and exploitation phases are present, with more variations at early stages of the runtime.



(a) Cortex-A7, Performance



(b) Cortex-A7, Power

Figure 8: Performance and Power Consumption of the generated RTM governor on Cortex-A7

Table 2 compares the average performance and power saving between the code generated RTM and the ondemand governor for the video decoding application on three platforms, e.g. for Cortex-A7 the generated RTM achieves 98% of the performance of the ondemand while using 61% of power used with the ondemand. It can be seen that across 3 different processors the generated RTM provides better power and energy savings while maintaining similar performance. The amount of saving varies with platforms due to the difference in number of VF pairs and in relationship between power, voltage and frequency. The power/energy factors are calculated by using system calls to measure time, and current is measured using a current sensor by setting the operating voltage. Performance is the percentage of deadlines that are passed (achieved). This is computed by dividing the number of frames with deadlines passed over the total frames processed.

Table 2: Power and performance result for code generated RTM compared to the ondemand, Video decoding application

Processor	**Performance (%)**	**Power (%)**	**Energy (%)**
Cortex-A8	99	95	96
Cortex-A7	98	61	62
Cortex-A15	96	77	80

Regarding the Jacobian matrix solver experiment on Cortex-A15, we achieved 100% of the performance of the ondemand while using 18% of power used with the ondemand. We experimented with the matrix solver at exactly 10 times a second. The results are different to the video decoder experiment, since ondemand chooses close to the maximum frequency whilst the RTM, monitoring the application throughput, recognises that the second lowest frequency is sufficient, i.e. 2000MHz vs 300Mhz.

The code generation that we used performs a fairly direct translation of the refined Event-B models to C so the generated algorithms will be as efficient in terms of complexity as the source Event-B model. We had one manually-written RTM implementation of the same RTM algorithm for the Cortex-A7 to compare against in terms of size and complexity. The number of lines of code for the generated code was less than the manually written code (1175 lines for the manually written code versus 475 for the generated code); the difference is largely down to coding style. Both implementations have similar algorithmic structure and thus similar algorithmic complexity.

6. Conclusion

We presented a model-based framework addressing complexity in RTM software programming due to the diversity of hardware platform characteristics. Although the designer needs to know the formal language and the associated toolset, the formal design model is built once and *specific* RTM software for different platforms is automatically generated from an *identical* formal design model. This can result in time saving compared to manual adjustment of the RTM implementation.

In addition to the automatic code generation, formal modelling is augmented by verification techniques. The correctness of the RTM design specifications and consistency of the refinement levels can be ensured by theorem proving and model checking.

We have validated our framework by applying it to develop an RL-based RTM system for a deadline-based application. The Event-B formal language is used to develop a *single* design model supporting platforms with different characteristics, and the RTM implementations are generated in the C programming language *specifically* for each platform.

We instantiated the RTM design model for three platforms with different characteristics and performed code generation for each of them; this is followed by evaluation of the effectiveness of the generated implementations in terms of power consumption. In all of the three experiments, energy saving is achieved compared to the Linux-ondemand governor.

To the best of our knowledge, this is the first reported investigation into automatic generation of embedded RTM and verification using high level model specification. The focus of this paper is evaluating the support for portability of RTM embedded across multiple hardware platforms. We envisage the framework working for wider experiments; In our ongoing work the Event-B models are being refined to support RTM algorithms for multi-core architectures and concurrent application.

Acknowledgment

This work was supported by an Engineering and Physical Sciences Research Council Programme Grant, EP/K034448/1 (See www.prime-project.org for more information).

References

[1] K. Choi, W.-C. Cheng, M. Pedram, Frame-based dynamic voltage and frequency scaling for a MPEG decoder, JOLPE 1 (1) (2005) 27–43. doi:10.1166/jolpe.2005.005.

[2] Y. Gu, S. Chakraborty, Control theory-based DVS for interactive 3D games, in: DAC, ACM Press, New York, New York, USA, p. 740. doi:10.1145/1391469.1391659.

[3] M. K. Bhatti, C. Belleudy, M. Auguin, Hybrid power management in real time embedded systems: an interplay of DVFS and DPM techniques, RTS 47 (2) (2011) 143–162. doi:10.1007/s11241-011-9116-y.

[4] S.-y. Bang, K. Bang, S. Yoon, E.-y. Chung, Run-time adaptive workload estimation for dynamic voltage scaling, IEEE TCAD 28 (9). doi:10.1109/TCAD.2009.2024706.

[5] M. Moeng, R. Melhem, Applying statistical machine learning to multicore voltage & frequency scaling, in: Proceedings of the 7th ACM international conference on Computing frontiers, ACM, New York, New York, USA, 2010, pp. 277–286. doi:10.1145/1787275.1787336.

[6] J. Abrial, Formal Methods: Theory Becoming Practice, J. UCS 13 (5) (2007) 619–628.

[7] J. Abrial, Modeling in Event-B - System and Software Engineering, Cambridge University Press, 2010.

[8] A. Edmunds, M. J. Butler, Linking Event-B and concurrent Object-Oriented programs, Electr. Notes Theor. Comput. Sci. 214 (2008) 159–182. doi:10.1016/j.entcs.2008.06.008.

[9] L. A. Maeda-Nunez, A. K. Das, R. A. Shafik, G. V. Merrett, B. Al-Hashimi, Pogo: an application-specific adaptive energy minimisation approach for embedded systems, in: HIPEAC Workshop on Energy Efficiency with Heterogeneous Computing, 2015.
URL HiPEAC

[10] A. Salehi Fathabadi, L. Alfonso Maeda-Nunez, M. Butler, B. Al-Hashimi, G. V. Merrett, Towards automatic code generation of run-time power management for embedded systems using formal methods (2015) 104–111doi:10.1109/MCSoC.2015.28.

[11] R. Sutton, A. Barto, Reinforcement learning: An introduction, Vol. 28, Cambridge Univ Press, 1998.

[12] G. E. P. Box, Understanding exponential smoothing: a Simple way to forecast sales and inventory, Quality Engineering.

[13] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466. doi:10.1007/s10009-010-0145-y.

[14] S. Hallerstede, On the purpose of Event-B proof obligations, in: ABZ, London, UK, 2008, pp. 125–138. doi:10.1007/978-3-540-87603-8_11.

[15] M. Leuschel, M. Butler, ProB: an automated analysis toolset for the B method, Software Tools for Technology Transfer (STTT) 10 (2) (2008) 185–203.

[16] A. Salehi Fathabadi, M. Butler, A. Rezazadeh, Language and tool support for event refinement structures in Event-B, Springer London, 2014, pp. 1–25. doi:10.1007/s00165-014-0311-1.

[17] A. Sinha, A. Chandrakasan, Dynamic voltage scheduling using adaptive

filtering of workload traces, in: VLSI Design, IEEE Comput. Soc, 2001, pp. 221–226. doi:10.1109/ICVD.2001.902664.

[18] S. Sinha, J. Suh, B. Bakkaloglu, Y. Cao, Workload-aware neuromorphic design of low-power supply voltage controller, IEEE JETCAS 1 (3) (2011) 381–390. doi:10.1109/JETCAS.2011.2165233.

[19] S. Yoo, A. A. Jerraya, Introduction to Hardware Abstraction Layers for SoC, in: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany, 2003, pp. 10336–10337. doi:10.1109/DATE.2003.10203.

[20] V. Pallipadi, A. Starikovskiy, The ondemand governor, in: Proceedings of the Linux Symposium, 2006.

[21] ARM, ARM Cortex-A8 Reference Manual (2010).