An Application- and Platform-agnostic Control and Monitoring Framework for Multicore Systems

Graeme M. Bragg[*], Charles Leech[*], Domenico Balsamo[*], James J. Davis[†],
Eduardo Wachter[*], Geoff V. Merrett[*], George A. Constantinides[†] and Bashir M. Al-Hashimi[*]
[*]School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK. Email: gmb@ecs.soton.ac.uk
[†]Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, UK

Abstract*—Heterogeneous multiprocessor systems have increased in complexity to provide both high performance and energy efficiency for a diverse range of applications. This motivates the need for a standard framework that enables the management, at runtime, of software applications executing on these processors. This paper proposes the first fully application- and platform-agnostic framework for runtime management approaches that control and optimise software applications and hardware resources. This is achieved by separating the system into three distinct layers connected by an API and cross-layer constructs called knobs and monitors. The proposed framework also supports the management of applications that are executing concurrently on heterogeneous platforms. The operation of the proposed framework is experimentally validated using a basic runtime controller and two heterogeneous platforms, to show how it is application- and platform-agnostic and easy to use. Furthermore, the management of concurrently executing applications through the framework is demonstrated. Finally, two recently reported runtime management approaches are implemented to demonstrate how the framework enables their operation and comparison. The energy and latency overheads introduced by the framework have been quantified and an open-source implementation has been released[1].

***Keywords*—Heterogeneous systems, runtime management, software framework.**

I. Introduction

The management and control of hardware settings at runtime has become a non-trivial task for multiprocessor embedded systems. In addition, applications have become increasingly dynamic to exploit the capabilities of these systems, with adjustable parameters that must be tuned to optimise their behaviour. As a result, the proactive optimisation of application performance and system energy efficiency is a key research challenge. Runtime management is a solution to this challenge that enables optimisation of, and tradeoff between, quality, application throughput and energy with varying requirements.

One way in which this can be achieved is by the exposure and adaptation of tunable parameters from the application and platform through a consistent framework interface. However, the majority of current frameworks only provide a mechanism to monitor the application's performance, and do not allow for the simultaneous monitoring and control of hardware components and applications at runtime. Moreover, most existing frameworks do not support heterogeneous platforms, which contain processors with differing capabilities, or the management of concurrent applications.

In this paper, the first framework for fully application- and platform-agnostic runtime management that enables simultaneous control and optimisation of software applications and hardware resources is presented. This is achieved by separating a system into three distinct layers of application, runtime management and device. These layers are connected through a novel API and cross-layer constructs called knobs and monitors, which enable the flow of information between the layers and the control and monitoring of runtime-tunable and -observable parameters. This reduces the design complexity by enabling the runtime management layer to provide a specific service to the applications, *e.g.*, to meet a performance requirement, whilst meeting optimisation targets by controlling the hardware resources. The novel contributions of the proposed framework are:

- the ability to control and monitor applications and hardware simultaneously using a novel cross-layered approach;
- a novel API that provides a consistent way in which knobs and monitors are specified and monitored across applications and platforms;
- a mechanism to enable the management of concurrently executing applications and heterogeneous platforms.

Additionally, the framework enables the direct comparison of different runtime management approaches and algorithms, which has not previously been possible, and simplifies runtime manager (RTM) development.

The framework is experimentally validated with a range of applications and two different types of heterogeneous platform to demonstrate its application- and platform-agnostic properties and to illustrate its ease of use. The management of two concurrently-executing applications is then demonstrated. In addition, two recently reported runtime management approaches, one based on control using performance counters and one that uses reinforcement learning, are implemented with the framework to demonstrate how the framework enables their operation and comparison. An open-source C++ implementation of the framework and API has also been released.[1]

This work was supported by the PRiME programme grant EP/K034448/1 (http://www.prime-project.org) and EPSRC grant EP/L000563/1.
Data supporting the results presented in this paper is available at ???DOI???.

[1]Available at: https://github.com/PRiME-project/PRiME-Framework

TABLE I
PROPERTIES OF STATE-OF-THE-ART FRAMEWORKS FOR RUNTIME MANAGEMENT OF APPLICATIONS ON MULTIPROCESSOR SYSTEMS.

Framework	Application–RTM				RTM–device		Monitor bounding	Heterogeneous platforms	Open-source release
	Knobs	Monitors	Non-temporal monitors	Multiple monitors	Knobs	Monitors			
Heartbeats [1]	✗	✓	✗	✗	✗	✗	✗	✗	✓
PowerDial [2]	✓	Heartbeats	✗	✗	✗	✗	✗	✗	✗
Heterogeneous Heartbeats [3]	✗	Heartbeats	✗	✗	✗	✓	✗	CPU + FPGA	✓
ARGO [4]	✓	✓	✓	✓	✗	✗	✗	✗	✗
AS-RTM [5]	✓	Heartbeats	✗	✓	✗	✗	✗	✗	✗
PTRADE [6]	✗	Heartbeats	✗	✗	✓	✗	✗	✗	✗
DRM [7]	✗	Heartbeats	✗	✗	✗	✗	✗	✗	✗
BEEPS [8]	✗	Heartbeats	✗	✗	✓	✗	✗	✗	✗
Proposed	✓	✓	✓	✓	✓	✓	✓	✓	✓

A survey of existing frameworks is presented in Section II. The proposed framework is explored in Section III and then practically validated in Section IV. Finally, Section V concludes the paper.

II. RELATED WORK

Various runtime management approaches exist in the literature for optimising system behaviour, whilst satisfying application requirements. These include dynamic voltage and frequency scaling (DVFS) [9], per-core power gating [10], dynamic task mapping and thread migration [11]. While RTMs are typically designed to address general challenges, such as energy efficiency or thermal management, they are largely implemented on specific platforms or with specific classes of application, *e.g.* multimedia [12] or image processing [13].

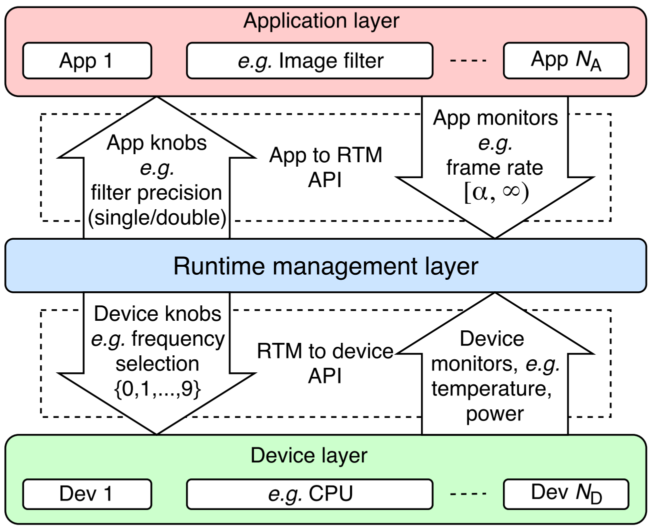
In addition, benchmarks are typically used to assess relative performance and measure specific aspects of RTMs and hardware platforms. However, they do not typically expose application requirements (*e.g.* error or accuracy) in addition to performance and this can limit the range of optimisation opportunities of runtime management approaches. Furthermore, source code for RTMs is often not released, with limited detail on implementation reported, making reproduction of results a non-trivial task. This prevents the direct comparison of approaches, with several works relying on indirect comparison via Linux governors [11], [14].

Runtime management can be enhanced by the exposure of dynamic knobs and monitors, which provide a mechanism to communicate with the application and platform. Specifically, knobs allow the tuning of hardware and application parameters by the RTM, while monitors enable the measurement of hardware properties and the observation of application behaviour, including the setting of performance targets by the application [2]–[4], [15]. In addition, knobs and monitors can been used to explore application-device tradeoffs, such as throughput-power [6] and precision-throughput [16], and locate optimal operating points for applications [17]. However, runtime management lacks portability unless these knobs and monitors are exposed through a consistent interface.

Several frameworks have been proposed in the literature to address the challenge of providing a consistent interface for runtime management. Table I summarises the features of these frameworks, showing whether application and device knobs and monitors are provided, as well as considering support for heterogeneous platforms. The most relevant framework is the Heartbeats API [1], which provides a standardised interface for single or concurrent applications to communicate their current and target performance to external observers, such as an RTM. However, the Heartbeats API only allows applications to communicate their throughput (*i.e.* the heart rate), therefore it does not allow other types of parameters to be exposed, such as accuracy and error (classed as non-temporal monitors in column 4 of Table I), and prevents tradeoffs between them. In addition, it does not extend this interface for monitoring or control of device parameters. Most of the frameworks reported in Table I are based on the Heartbeats concept and inherit its features, *e.g.* application monitors (column 3).

In order to perform tradeoffs within a single application, multiple monitors of different types must be exposed, *e.g.* throughput and error. Column 5 of Table I shows that Heartbeats, and most of the frameworks that rely on it, do not support this functionality. In addition, for an application to meet its requirements, a target can be specified with the monitor. However, there is no indication as to whether the target is a maximisation or minimisation objective, as listed in column 8. As a result, these approaches do not allow fully application-agnostic behaviour.

Columns 6 and 7 show that current frameworks only provide partial abstraction of RTM to device communication, and do not include both knobs and monitors to control hardware components at runtime. Moreover, most existing works do not operate on heterogeneous platforms (column 9), which provide both high performance and energy efficiency by combining conventional CPUs with other accelerators. These platforms



Application layer
App 1
e.g. Image filter
App N_A
App knobs e.g. filter precision (single/double)
App to RTM API
App monitors e.g. frame rate $[\alpha, \infty)$
Runtime management layer
Device knobs e.g. frequency selection {0,1,...,9}
RTM to device API
Device monitors, e.g. temperature, power
Device layer
Dev 1
e.g. CPU
Dev N_D

Fig. 1. Cross-layer framework and API enabling communication between the application, runtime management and device layers using knobs and monitors. Examples are given for an image filter application on a CPU.

typically increase the scalability of parallel applications and systems, and therefore they need to be managed by a framework that supports device-agnostic control. The only approach that is based on a heterogeneous platform introduces a hardware dependency in the process [3]. This restricts the cross-platform capabilities of current frameworks, meaning that they do not allow current RTM approaches to be portable across multiple platforms.

III. Proposed Framework

To address the limitations of existing frameworks discussed in Section II, a framework for application- and platform-agnostic runtime management of heterogeneous systems is presented. Fig. 1 shows the proposed framework and how the three layers are connected by novel APIs (`App to RTM API` and `RTM to device API`). This provides consistent interfaces from an RTM to both hardware platforms and applications, which enables the design and implementation of application- and platform-agnostic runtime management approaches. As discussed in Section II, application knobs expose tunable application parameters, *e.g.* filter precision, while monitors convey information about the behaviour of the applications, *e.g.* frame rate. Similarly, device knobs expose tunable device parameters while monitors convey information about the status of devices. Exposing knobs and monitors at both the application and device layer enable tradeoffs, *e.g.* performance-energy or accuracy-temperature, to be explored and exploited by the runtime management layer.

In addition, the proposed framework facilitates the comparison of existing RTMs as well as the management of concurrently-executing applications and heterogeneous platforms. The remainder of this section provides an overview of the technical concepts of the proposed framework and details of the novel API.

A. Framework Concepts

Structure: The separation of the system into three distinct layers, application, runtime management and device, as illustrated in Fig. 1, reduces design complexity and provides flexibility during operation. Specifically, the application layer comprises any number of software processes, while the device layer includes the hardware and its software drivers. The runtime management layer comprises an RTM responsible for the control and monitoring of the other two layers. This separation ensures portability and cross-compatibility; applications and device drivers only need to be written once to be used with any implemented RTM.

Communication: Knobs and monitors, shown in the dashed regions of Fig. 1, facilitate communication between the layers. Bounds are attached to both knobs and monitors, in the form of *minima* and *maxima*, which allow applications and devices to inform the runtime management process of targets and constraints. Knob bounds represent a range of *allowed* values while monitor bounds represent a range of *desired* values rather than a single target value. An RTM's primary objective is to ensure that the monitor values of all applications and the device remain within their specified bounds. Minimal modification is required to applications to expose knobs and monitors through the framework.

For example, the image filter application shown in Fig. 1 provides the option of selecting float or double precision for its numeric operations at runtime. Under the proposed framework, this choice would be controlled by the RTM using an application knob with options $\{0, 1\}$. If the same application requires a minimum throughput, *e.g.*, expressed as a frame rate α, an application monitor with this minimum bound can be provided to the RTM. In this case, the application periodically updates the current frame rate so that the RTM can keep it within the range $[\alpha, \infty)$. On the hardware side, the example of a CPU is considered within the device layer (Fig. 1). DVFS of the CPU is achieved *via* a device knob with options $\{0, 1, \cdots, 9\}$, enabling the RTM to switch between ten distinct voltage-frequency pairs. Finally, to enable thermal management by the RTM, a temperature sensor is considered as an illustrative device monitor.

Weights: Individual applications may feature multiple performance objectives with differing priorities. For example, an application aware of both its throughput and accuracy may wish to prioritise the optimisation of one over the other. In the proposed framework, this priority is expressed with a numeric weight attached to each monitor. These weights instruct the RTM to expend proportional effort in optimising each monitor's value. In a similar manner, application priority is indicated through an attached weight such that a higher level of consistency can be ensured by foreground processes in a multi-tasking scenario.

Concurrency: Real-world systems commonly execute more than one application concurrently that compete for hardware resources. Due to this, the runtime manager is required to carefully manage system resources so that each application

TABLE II
APPLICATION-TO-RTM AND RTM-TO-DEVICE API FUNCTIONS FOR THE PROPOSED FRAMEWORK.

Layer	Construct	Space	Identifier	Input(s)	Output(s)	Description
app	knob	disc / cont	min	knob, min	–	Update application knob's minimum allowed value
			max	knob, max	–	Update application knob's maximum allowed value
			get	knob	value	Pull application knob's current value
	mon		min	mon, min	–	Update application monitor's minimum desired value
			max	mon, max	–	Update application monitor's maximum desired value
			weight	mon, weight	–	Update application monitor's relative importance
			set	mon, value	–	Push application monitor's current value
dev	knob		min	knob	min	Pull device knob's minimum allowed value
			max	knob	max	Pull device knob's maximum allowed value
			init	knob	init	Pull device knob's initial (default) value
			type	knob	type	Pull device knob's type
			set	knob, value	–	Push device knob's current value
	mon		type	mon	type	Pull device monitor's type
			get	mon	value	Pull device monitor's current value and bounds

meets its performance targets. When considering concurrently-executing applications, the framework provides a mechanism to identify and manage them simultaneously using a unique ID number. In this way, the knobs and monitors can be grouped and traded-off between applications by the RTM.

Types: Knobs and monitors each have a *type* selectable from a discrete set of options, *e.g.*, TEMP for a temperature monitor or FREQ for a frequency knob. This represents a compromise between complete agnosticism and the full provision of information. Providing "hints" to the RTM simplifies the process of determining the function of knobs and the properties represented by monitors, *e.g.*, "lower power is better".

Spaces: All knobs and monitors are expressed in standardised, unit-less formats to maintain application and device agnosticism. The proposed framework allows discrete- and continuous-valued versions of each knob and monitor, so that the appropriate optimisation and control process can be used by the RTM. For example, a boolean choice (*i.e.* $\{0, 1\}$) does not require iterative convergence. These spaces enable the translation of application-specific information into agnostic sets, as shown in Fig. 1 for the ranges of the knobs and monitors. Discrete versions use signed integer values while their continuous counterparts operate using floating-point data.

Adaptability: In order to provide maximal flexibility, all bounds and weights are adjustable at runtime, and no restrictions are placed on when update to these can occur. Most commonly, applications create their knobs and monitors before being executed, however no limitation is imposed on such events occurring partway through application execution. Applications are allowed to be attached to and detached from the framework at any point during runtime. This capability is in contrast to existing frameworks, most of which assume a constant application set, contrary to the typical use of many commercial embedded systems.

B. Framework API Specification

The proposed framework is realised through novel application-to-RTM and RTM-to-device APIs, which connect the system layers of Fig. 1 and enable the exposure of knobs and monitors between the three layers in a consistent manner across different applications and hardware platforms. Table II illustrates how the API functions are split into application (app) and device (dev) categories, with subcategories for knob (knob) and monitor (mon) interaction. Discrete- (disc) and continuous-valued (cont) versions exist across the API to indicate the typology of the knob or monitor being used.

The RTM must be made aware of the allowable and desired values for knobs and monitors, respectively, in order to ensure that its optimisation targets are regulated and have positive effects. For knobs, functions app_knob_(disc|cont)_(min|max)() facilitate this, letting the application indicate the range in which values can be chosen. Conversely, monitor functions app_mon_(disc|cont)_(min|max|weight)() allow the setting of RTM objectives, with *_min() and *_max() functions indicating desired lower and upper bounds. Where an application requires only a maximum or minimum bound, the other end of the range can be left unbounded using (DISC|CONT)_MIN or (DISC|CONT)_MAX. Intra-application weighting values between 0.0 and CONT_MAX can be used to indicate relative monitor importance to the RTM using *_weight() functions, guiding its optimisations. All of these settings can be updated during application execution if required. Functions app_knob_(disc|cont)_get() and app_mon_(disc|cont)_set() are used by the application to get the current value of a knob from the RTM and set an updated value for a particular monitor to the RTM, respectively.

Device-layer knobs and monitors are exposed and updated *via* the RTM-to-device API functions, as shown in the lower half of Table II. Functions dev_knob_(disc|cont)_(min|max)() are equivalent to their application-layer counterparts, setting the range of valid values. Additional functions dev_knob_(disc|cont)_(type|init)() return the type of the knob or its initial value (*i.e.* the default value). Type-related functions return values from defined sets and are called by the RTM using dev_mon_(disc|cont) `_type()`. The RTM uses functions `dev_knob_(disc|cont)_set()` and `dev_mon_(disc|cont)_get()` for setting device knob values and accessing monitor values and bounds from the device at runtime.

An open-source C++ implementation of the framework and API has been released[1]

IV. Evaluation

In order to demonstrate the capabilities of the framework and validate its operation, a series of experiments have been carried out. Illustrative example runtime controllers were used where appropriate to demonstrate specific concepts. The experimental setup is discussed in Section IV-A. The basic operation of the framework and its ease-of-use are demonstrated in Section IV-B with an illustrative runtime controller. Application agnosticism is shown throughout this section while platform agnosticism is demonstrated in Section IV-C with the same application and RTM executing on two different heterogeneous platforms. Support for concurrent applications is validated in Section IV-D, with two different applications executing on one platform. Additionally, the ability of the framework to enable direct comparison of RTMs is shown in Section IV-E with two recently reported runtime management approaches. Finally, overheads are analysed in Section IV-F.

TABLE III
Device-level knobs and monitors for the Odroid-XU3.

Const.	Space	Type	For	No.
knob	`disc`	`FREQ`	LITTLE cluster	1
	`disc`	`FREQ`	big cluster	1
	`disc`	`FREQ`	GPU	1
mon	`cont`	`POW`	Clusters, RAM, GPU, SoC	5
	`cont`	`TEMP`	big cores	4
	`cont`	`TEMP`	GPU	1
	`disc`	`PMC`	LITTLE cores	16
	`disc`	`PMC`	big cores	24

Listing 1
RTM code for agnostic control and monitoring of application and device knobs and monitors.

```
 1 void rtm::control_loop(){
 2  while(1){
 3   temp_mon = dev_api.mon_cont_get(temp_mons[2]);
 4   if(apps.size()){
 5    app_perf = app_mons_cont[0];
 6    if(app_perf.val < app_perf.min){
 7     if(freq_knob.val < freq_knob.max){
 8      freq_knob.val++;
 9      dev_api.knob_disc_set(freq_knob, freq_knob.val);
10    }}
11    else if(temp_mon.val > temp_mon.max){
12     freq_knob.val--;
13     dev_api.knob_disc_set(freq_knob, freq_knob.val);
14 }}}}
```

A. Experimental Setup

Two heterogeneous embedded platforms were used to demonstrate the proposed framework. The Odroid-XU3 development board, containing an ARM big.LITTLE design with two quad-core CPU clusters and a GPU, was used to demonstrate the ease-of-use of the framework, the direct comparison of RTMs and to assess overheads. The platform contains five temperature sensors to monitor the CPU and GPU, and four power sensors to monitor each CPU cluster, the GPU and memory. Each of these is exposed to the framework as a device monitor. Three device knobs are exposed to provide DVFS for each CPU cluster and the GPU. Table III summarises the knobs and monitors of the Odroid-XU3.

A second platform, the Cyclone V SoC Development Kit, was used to demonstrate platform-agnostic operation of the framework. This platform includes a heterogeneous CPU-FPGA system-on-chip containing two ARM CPUs and FPGA fabric. Using OpenCL, applications can execute on either the CPUs or the FPGA as synthesised hardware.

Four different applications from the numerical and multimedia domains were used to demonstrate the application-agnostic properties of the framework.

B. Agnostic Runtime Management

A basic runtime controller was implemented for the RTM layer to illustrate the use of knobs and monitors for maintaining an application performance target while optimising a given device monitor. Listing 1 shows the code for the controller, which ensures that the value of the application performance monitor remains within its bounds. This is achieved by adjusting the device frequency knob in order to avoid violations of the monitor bounds `app_perf.min` and `app_perf.max` (lines 6 – 9). The optimisation of device temperature is the secondary objective of the controller (by monitoring the current value `temp_mon.val`) and is achieved by decrementing the frequency knob (line 12), trading-off excess application performance (lines 12 – 13). The `else if` statement on line 11 ensures that the performance monitor is prioritised over the device temperature.

The behaviour of this controller is shown in Fig. 2 while running a numerical benchmarking application (Whetstone). This benchmark performs numerical functions using integer and floating-point arithmetic. Its performance is measured in thousands of Whetstone instructions per second (KIPS), which is exposed as a continuous monitor with bounds of $[2.30, \infty)$. Initially, the controller set the device frequency to maximum and observed the device temperature. As the temperature increased above the the maximum threshold specified by `temp_mon.max` (80°C), the controller reduced the frequency until the temperature was below the threshold whilst ensuring that the application performance was higher than `app_perf.min`. After 50 seconds, the platform reduced the temperature threshold to 60°C and the RTM reduced the frequency in response until the updated monitor bound was met while exceding the application throughput requirement.

This experiment demonstrates the basic operation of the framework and illustrates the dynamic nature of knobs and monitors. The controller is application- and platform-agnostic as it could operate, without modification, with any application that exposes a performance monitor and with any platform that exposes a frequency knob and temperature monitor.

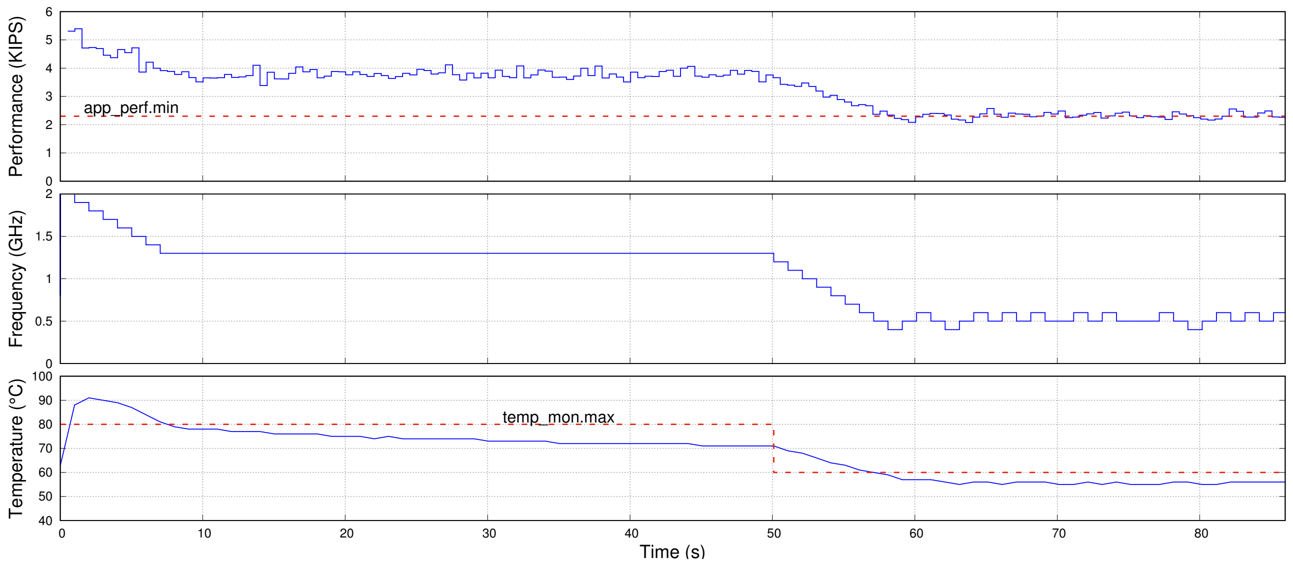


Fig. 2. Device temperature optimisation under application performance constraints using the controller RTM, including dynamic adjustment of the temperature threshold from 80 to 60°C.

C. Platform Agnosticism

The portability of RTMs and applications implemented within the framework is demonstrated in Fig. 3, which shows the design-space exploration (DSE) of the same application across two heterogeneous platforms using the same RTM code. A Jacobi iterative solver benchmark was used as a case-study application. It can operate a tradeoff between the speed of calculation (solves per second) and the accuracy of the result (mean squared error) by adjusting the number of iterations performed and the precision of the data type. Throughput and accuracy are exposed as monitors while iterations and the precision are exposed as knobs. The DSE extended to application execution on the heterogeneous components of both platforms, including the GPU on the Odroid and the FPGA on the Cyclone V, in addition to the CPUs. Points in Fig. 3 show the resultant throughput and error for each combination of knob values with blue crosses for the Odroid and green triangles for the Cyclone V. This experiment demonstrates that the same application and RTM code can be used on any platform supported within the proposed framework.

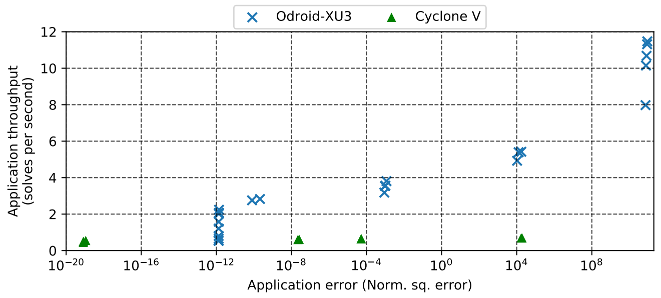


Fig. 3. Design-space exploration of the Jacobi application across the Odroid-XU3 and Cyclone V devices.

D. Concurrent Application Management

This subsection demonstrates how the framework supports the management of concurrently executing applications. A runtime control algorithm was implemented with a target of keeping the throughput monitor of each application within its bounds, `app_perf.min` and `app_perf.max`, while minimising device frequency. The behaviour of this controller is shown in Fig. 4, where the execution of two applications is indicated by their throughput over time. The top plot shows a video filtering application and the middle plot shows the Jacobi iterative solver.

Initially, the video filter application was the only application executing. As a result, the runtime controller adjusted the CPU frequency to meet the application throughput bounds at the lowest frequency possible. The Jacobi application began its execution at 21 seconds and reported that its throughput was below the minimum bound to the RTM. In addition, the throughput of the video filter decreased due to competition for device resources. To compensate, the controller increases the CPU frequency such that the throughput of both applications was within their bounds.

E. Comparison of RTM approaches

To demonstrate the framework's optimisation and comparative capabilities, two state-of-the-art approaches were implemented within the proposed framework. The first approach, RTM-A [11], aims to optimise power consumption by monitoring hardware performance counters to adjust CPU frequency. The second approach, RTM-B [18], employs reinforcement learning to predict the frequency that should be selected to meet an application performance target. RTM-A was originally evaluated on the Odroid-XU3 platform, using standard benchmarks, reporting a mean energy saving of 25% when

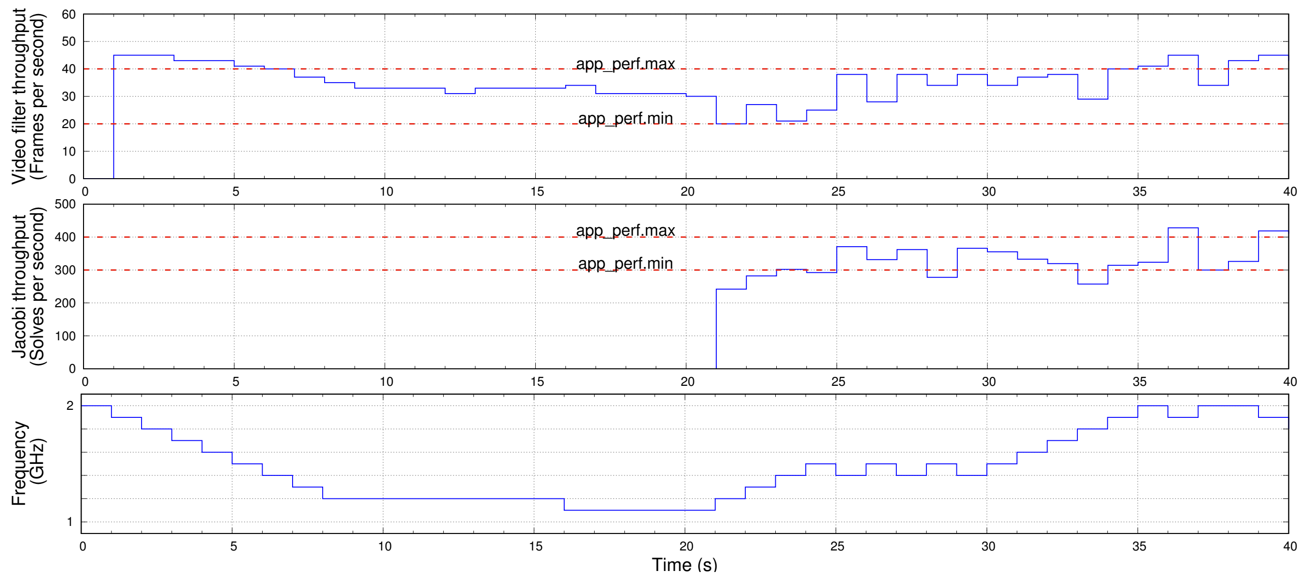


Fig. 4. Runtime management of the throughput of two concurrently-executing applications through the framework. The Jacobi application begins execution at 21 seconds and the device frequency is adjusted to compensate.

compared to the Linux Ondemand governor. RTM-B was evaluated on the BeagleBoard-xM platform, using a video decoder application, reporting a mean reduction in energy consumption of 30% when compared to Ondemand.

These two approaches lack portability and direct comparisons cannot be made, due to the different platforms used for experimental validation. Implementation within the proposed framework allows them to be directly compared, saving development time and improving the accuracy of the comparison. To demonstrate how this comparison can be made, the RTMs were evaluated using a OpenCV video decoding application on the Odroid-XU3 platform. The application exposes a continuous monitor for the frame rate, with a minimum bound of 25 frames per second. The RTMs are directly compared in Fig. 5, between bars 2 and 4, showing that the application consumed a mean total energy of 381 J and 376 J under the control of RTM-A and RTM-B, respectively. Comparison with the Linux Ondemand governor (bar 5) shows energy savings of 17.2% and 18.2%, respectively. This demonstrates that while RTM-B achieves a greater energy saving, it is less than reported in literature for this application and platform pair.

F. Overheads

As with any abstraction, the framework introduces an energy overhead due to the additional computation required. This overhead can be estimated by comparing stand-alone versions of RTM-A and RTM-B against their implementations within the framework. Results of these experiments can be seen in Fig. 5 for RTM-A (bars 1 and 2) and for RTM-B (bars 3 and 4). RTM-A required 19.6 J (5.48%) more energy, while RTM-B required only 15.2 J (4.23%) more energy, in the minimum case. The minimum case was used to minimise the impact of other running processes on the result. When compared to the

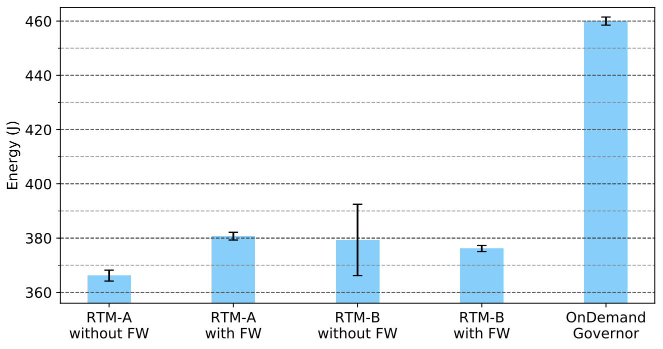


Fig. 5. Mean total energy consumed by the Odroid-XU3 running the video decoder application under the control of each RTM, both with and without the framework (FW). The experiment was repeated 50 times for each RTM.

Ondemand governor, the two RTMs still achieved significant savings despite these overheads.

The framework also introduces latency overheads that limit the response rate of the RTM. Fig. 6 visualises the steps involved in reading a device monitor inside the framework, identifying seven internal latency sources. t_{asm}, t_{tx} and t_{diss} are the times to assemble, transmit and disassemble the message, t_{net} is the message-passing interface latency and t_{search} is the time to search for and read a monitor. The latency related to each API call was measured and found to be 80–200 μs, with 40% attributed to cross-layer communication. For an RTM reading one device monitor and setting one device knob per update, this limits the update rate to 1.67 kHz in the worst case.

V. Conclusions

This paper has presented a framework that enables application- and platform-agnostic runtime management of

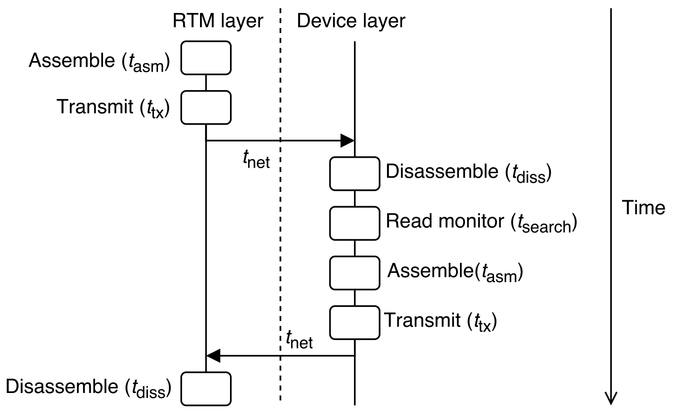


Fig. 6. Breakdown of the sources of latency introduced by the framework for communication between the RTM and device layers.

concurrently executing applications on heterogeneous multi-core systems. This is achieved by visualising a system as three distinct layers that are connected by dynamic knobs and monitors that allow a range of tunable parameters and observable metrics to be exposed. The framework enables the direct comparison of different RTM approaches, which has not been previously possible, and simplifies RTM development. The framework introduced very modest energy and latency overheads that have limited impact on the operation and performance of RTMs. Operation with concurrent applications was demonstrated and an open-source implementation of the framework has been released.[1] Research is ongoing to provide further validation of the framework and to integrate additional applications, devices and RTMs.

References

[1] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. I. I. I. I. Agarwal, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *Int'l Conf. on Autonomic Comput.*, 2010.

[2] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal *et al.*, "Dynamic Knobs for Responsive Power-aware Computing," in *Int' Conf. on Arch. Supp. for Prog. Lang. and OS*, 2011.

[3] S. T. Fleming and D. B. Thomas, "Heterogeneous Heartbeats: A Framework for Dynamic Management of Autonomous SoCs," in *Int'l Conf. on Field Prog. Logic and Appl.*, 2014.

[4] D. Gadioli, G. Palermo, and C. Silvano, "Application Autotuning to Support Runtime Adaptivity in Multicore Architectures," in *Int'l Conf. on Embedded Comput. Syst.: Architectures, Modeling, and Simulation*, 2015.

[5] E. Paone, D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano, "Evaluating Orthogonality between Application Auto-tuning and Run-time Resource Management for Adaptive OpenCL Applications," in *IEEE Int'l Conf. on Appl.-specific Syst., Arch. and Proc.*, 2014.

[6] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "A Generalized Software Framework for Accurate and Efficient Management of Performance Goals," in *Int'l Conf. on Embedded Software*, 2013.

[7] A. Baldassari, C. Bolchini, and A. Miele, "A Dynamic Reliability Management Framework for Heterogeneous Multicore Systems," in *IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Syst.*, 2017.

[8] F. Gaspar, L. Taniça, P. Tomás, A. Ilic, and L. Sousa, "A Framework for Application-guided Task Management on Heterogeneous Embedded Systems," *ACM Trans. on Arch. and Code Optim.*, vol. 12, no. 4, 2015.

[9] A. Das, R. A. Shafik, G. V. Merrett, B. M. Al-Hashimi, A. Kumar *et al.*, "Reinforcement Learning-based Inter- and Intra-application Thermal Optimization for Lifetime Improvement of Multicore Systems," in *ACM/EDAC/IEEE Design Automation Conf.*, 2014.

[10] A. M. Rahmani, M. H. Haghbayan, A. Miele, P. Liljeberg, A. Jantsch *et al.*, "Reliability-aware runtime power management for many-core systems in the dark silicon era," *IEEE Trans. on VLSI Syst.*, vol. 25, no. 2, 2017.

[11] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster Thread-to-core Mapping and DVFS on Heterogeneous Multi-cores," *IEEE Trans. on Multi-Scale Comput. Syst.*, vol. PP, no. 99, pp. 1–1, 2017.

[12] Y. G. Kim, M. Kim, and S. W. Chung, "Enhancing Energy Efficiency of Multimedia Applications in Heterogeneous Mobile Multi-core Processors," *IEEE Transactions on Computers*, vol. 66, no. 11, 2017.

[13] S. Yang, R. A. Shafik, G. V. Merrett, E. Stott, J. M. Levine *et al.*, "Adaptive Energy Minimization of Embedded Heterogeneous Systems using Regression-based Learning," in *Int'l Workshop on Power and Timing Modeling, Optim. and Sim.*, 2015.

[14] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *2015 Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2015.

[15] C. Leech, C. Kumar, A. Acharyya, S. Yang, G. V. Merrett *et al.*, "Runtime performance and power optimization of parallel disparity estimation on many-core platforms," *ACM Trans. on Embedded Comput. Syst. (TECS)*, vol. 17, no. 2, p. 41, 2018.

[16] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive Control of Approximate Programs," in *Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, 2016.

[17] V. Vassiliadis, C. Chalios, K. Parasyris, C. D. Antonopoulos, S. Lalis *et al.*, "Exploiting Significance of Computations for Energy-constrained Approximate Computing," *Int'l J. of Parallel Prog..*, vol. 44, no. 5, 2016.

[18] 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 Heterogenous Comput.*, 2015.