Model-free Runtime Management of Concurrent Workloads for Energy-Efficient Many-Core Heterogeneous Systems

Ali Aalsaud\textsuperscript{1,3}, Ashur Rafiev\textsuperscript{2}, Fei Xia\textsuperscript{1}, Rishad Shafik\textsuperscript{1} and Alex Yakovlev\textsuperscript{1}
\textsuperscript{1} School of Engineering, \textsuperscript{2}School of CS University of Newcastle, Newcastle upon Tyne, NE1 7RU, England, UK \textsuperscript{3} School of Engineering, Al-Mustansiriya University, Baghdad, Iraq
\textsuperscript{1,2} \{ A.m.m.aalsaud, ashur.rafiev, Fei.Xia, Rishad.Shafik, Alex.yakovlev \}@ncl.ac.uk
\textsuperscript{3} a.m.m.aalsaud@uomustansiriyah.edu.iq

Abstract—Modern embedded systems execute multiple applications, both sequentially and concurrently, on heterogeneous platforms. Determining the most energy-efficient system configuration (i.e., the number of parallel threads, their core allocations and operating frequencies) tailored for each kind of workload is extremely challenging. In this paper, we propose a novel runtime optimization approach with the aim of maximizing power-normalized performance considering dynamic workload variations. To reduce overhead and complexity, we adopt a model-free approach to runtime adaptation based on workload classification. This classification is supported by analysis of data collected from a comprehensive study investigating the tradeoffs between inter-application concurrency with performance and power under different system configurations. We conduct extensive experiments on an Odroid XU3 heterogeneous platform with synthetic and standard benchmark applications to develop the control policies and validate our approach. These experiments show that workload classification into CPU-intensive and memory-intensive types provides the foundation for scalable energy minimization with low complexity. Implementing this approach as a Linux runtime governor, we demonstrate that IPS/Watt can be improved by over 120\% compared to existing approaches.

keywords—many-core systems; concurrent applications; runtime optimization; power-normalized performance; workload classification

I. INTRODUCTION

Contemporary computing systems, including embedded and high performance systems, are exhibiting increased complexities in two dimensions. In one dimension, the number and type of computing resources (cores) are growing in hardware platforms, and in the other, an increasing diversity of applications are being executed concurrently on these platforms \cite{1} \cite{2} \cite{3}. Managing hardware resources to achieve energy efficiency, under different application scenarios (single or concurrent), is proving highly challenging due to runtime space expansion \cite{4}.

As energy consumption becomes a limiting factor to continued technology scaling and performance improvements \cite{5}, techniques for increasing energy efficiency have emerged. To provide control over power/performance tradeoffs, dynamic voltage frequency scaling (DVFS) is integrated into contemporary devices, e.g., current Intel and ARM processors \cite{6}. DVFS suitably scales voltage/frequency across a number of predetermined operating points. These have different impacts on performance and power consumption and hence their choices need to be made based on the application workload. Another technique for improving energy efficiency is the parallelization of workloads \cite{3}, including suitable task mapping (TM) to cores.

DVFS and TM may be synergistically controlled at the system software level for effective energy optimization. For instance, DVFS is controlled in Linux with power governors \cite{7}, such as ondemand, performance, conservative, userspace and powersave. These governors use pre-set voltage/frequency points to manage system power according to the knowledge and prediction of workload and user preference. Current Linux governors are, however, not able to optimize energy consumption efficiently, primarily because they are unable to couple DVFS and dynamic TM \cite{7}. Further, these approaches, although serviceable, are not capable of taking advantage of the different degrees of parallelizability of individual applications that are typically seen in modern computing systems.

Mapping threads to cores (TM) is usually handled by a separate routine in the system software, for example the Linux scheduler \cite{8}. The scheduler seeks to spread the workload of all applications across multiple available cores to achieve maximum utilization. This approach is functional but leaves rooms for improvement. For instance, there is no discrimination about the thread workload type when being scheduled \cite{8}, such as CPU-intensive or memory-intensive. Not taking the workload type into account results in indiscriminate sub-optimization in power and performance, leading to poor energy efficiency \cite{9}\cite{10}.

![Table 1: Features of Existing Approaches and This Work](image)

Existing approaches can be categorized into two types:
offline (OL) and runtime (RT). In OL approaches, the system is extensively reasoned to derive energy and performance models [15][9][10]. In RT approaches, the models are typically learnt using monitored information [13][14][16]. Since RT modelling is costly in terms of resources, often OL and RT are complementarily coupled [9][16].

A number of approaches have been proposed over the years that consider energy optimization using OL, RT or a combination of both (see Table I). A recurring scheme in these approaches is that the energy efficiency is primarily focused on single-application workloads without considering its variations among concurrent applications. However, the same application can exhibit different energy/performance trade-offs depending on whether it is running alone or concurrently with other workloads. This is because: a) the workload context switches within the application between memory- and CPU-intensive contexts, and b) architectural sharing between applications affect the energy/performance trade-offs (see Section VI.B).

In this work, we develop an RT adaptation approach to improve the energy efficiency of a heterogeneous many-core system with concurrent workloads. Core to our approach is an empirical and data-driven method, which classifies applications based on their memory and CPU requirements. The aim is to derive DVFS and TM policies, tailored to the classified workloads without requiring any explicit modelling at RT. Due to simplified RT classification, our approach can significantly reduce overheads. Further, our model-free classification based RT enhances scalability for any concurrent application mix, platform, and metric having linear complexity which is not affected by the system heterogeneity, the number of concurrent applications. In comparison, linear complexity was achieved in existing work when dealing with single applications running on homogeneous systems using either of TM and DVFS (see Table I and Section VI.C for details). Otherwise they display combinatorial polynomial (P) or non-polynomial (NP) complexities in concurrent application scenarios. In this context, workload classification means the classification of each application into a workload taxonomy based on differences in processing and memory requirements.

A. Contributions

This paper makes the following specific contributions:

1) using empirical observations and CPU performance counters, derive RT workload classification thresholds, expressed in terms of instructions per cycle (IPC);
2) underpinned by the workload classification, propose a low-complexity, model free and low-cost RT approach for synergistic controls of DVFS and TM;
3) using synthetic and real-world benchmark applications with different concurrent combinations, investigate the approaches energy efficiency, measured by power-normalized performance in instructions per second (IPS) per Watt (IPS/Watt), i.e. instructions per Joule;
4) implement the approach as a Linux power governor and validate through extensive experimentation with significant IPS/Watt improvements.

To the best of our knowledge, this is the first work that uses workload classification (WLC) during RT to optimize both DVFS and TM for concurrent workloads on many-core heterogeneous platforms without requiring application instrumentation (see Table I).

II. RELATED WORK

A power control approach for many-core processors executing single applications was proposed in [17]. Among others, Goraczko et al. [11] and Luo et al. [12] proposed DVFS approaches with software task partitioning and mapping of single applications using a linear programming-based optimization during runtime to minimize the power consumption. Goh et al. [18] proposed a similar approach of task mapping and scheduling for single applications described by synthetic task graphs.

Other works have dealt with power minimization on heterogeneous platforms. For example, Yang et al. [13] presented an adaptive power minimization approach using runtime linear regression-based modeling of the power and performance tradeoffs. Using the model, the task mapping and DVFS are suitably chosen to meet the specified performance requirements. Nabina and Nunez-Yanez [14] presented a similar DVFS approach for FPGA-based video motion compensation engines using runtime measurements of the underlying hardware.

A number of studies have also made use of simulation tools like gem5, together with McPAT [15], [19] for single applications. These works have used DVFS, task mapping, and offline optimization approaches to minimize the power consumption for varying workloads.

Energy efficiency improvement approaches have also considered a single-metric based optimization: primarily performance-constrained power minimization, or performance improvement within a power budget [20].

In order to optimize some metric, the controller must have some means to calculate DVFS and TM decisions based on information from the execution. The control methods can be model-based with online learning of the model [21]. It may also involve some form of regression-based methods [13] or classical optimization techniques [18]. An analytical model can help the finding of the optimal operating configurations for the workload and system states. However the runtime acquisition and tuning of the model require overheads.

A model-free RT WLC approach with corresponding DVFS controls is proposed by Wang and Pedram [16]. This approach employs reinforcement learning, with the action space size a big concern for the authors, even though for only homogeneous systems at much higher granularities than CPU cores. WLC has also been used OL, but this produces a fixed class for each application [9], [10] and cannot deal with workload behaviour changes during execution.

III. SYSTEM PLATFORM AND APPLICATIONS

Our experimental investigations use a many-core platform to illustrate the suitability of the proposed approach, when
executing workloads on a number of heterogeneous cores. Further, we study scalability by executing a number of concurrent applications this example platform.

The platform of choice is the Odroid XU3 [22], which includes an SoC based on the ARM big.LITTLE architecture. It has eight general processing ARM Cortex cores. Four of these are low-power A7 cores and the other four high-performance A15 cores. Each group of four cores of the same type constitutes a power domain, which is supplied with the same frequency and voltage, and the XU3 provides RT power monitoring per power domain, and per-domain DVFS.

The A7 and A15 processor architectures also provide performance counters that record, per-core, instructions executed and clock active and idle cycles. This work uses the set of performance counters listed in Table II.

<table>
<thead>
<tr>
<th>TABLE II: PERFORMANCE COUNTER EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance counter</td>
</tr>
<tr>
<td>InstRet</td>
</tr>
<tr>
<td>Cycles</td>
</tr>
<tr>
<td>Mem</td>
</tr>
</tbody>
</table>

In our investigation, we chose a number of different applications. A synthetic benchmark, called psync, is developed, based on purely CPU-intensive stress enhanced with tunable memory access M, that is in linear relation to the real memory to computation ratio, to investigate the general CPU vs memory effects. In addition, a group of realistic application benchmarks from the PARSEC suite [23] is also included to span the range of CPU, memory, and mixed execution characteristics. Specifically, we chose the application ferret to represent CPU-intensive, fluidanimate to represent memory-intensive, and bodytrack to represent both CPU- and memory-intensive applications. It will be demonstrated later that psync is needed to represent pure CPU-only and memory-only tests because realistic applications, such as PARSEC benchmarks, have CPU- and memory-intensive contexts during their execution traces. This sets up one of the major motivations for classifying during RT.

IV. WORKLOAD CLASSIFICATION TAXONOMY

The taxonomy of workload classes chosen for this work reflect differentiation between CPU-intensive and memory-intensive workloads, with high- or low-activity. Specifically, workloads are classified into the following four classes:

- Class 0: low-activity workloads
- Class 1: CPU-intensive workloads
- Class 2: CPU- and memory-intensive workloads
- Class 3: memory-intensive workloads

Extensive explorative experiments are run in this work to investigate the validity of these general concepts. For instance, Figure 1 shows the energy efficiency of psync running on 2-4 A7 cores (one of the A7 cores was reserved for the operating system in these experiments, hence the data does not cover the single core case) with M values ranging from 0 to 1. It can be seen that with memory-intensive tasks (larger M), it is better to use fewer cores, but with CPU-intensive tasks (smaller M), it is better to run more cores in parallel. This and other results sweeping through the frequency ranges and core combinations with psync confirm the validity of the classification taxonomy and establish a TM and DVFS strategy based on relative CPU and memory use rates. The full set of psync experimental data, supported by experiments with applications other than psync, is used to generate our runtime management (RTM) presented in subsequent sections.

V. RUNTIME MANAGEMENT AND GOVERNOR DESIGN

Figure 2 presents the general architecture of RTM inside a system. In this section we explain the central RTM functions classification and control actions based on performance monitors and actuators (e.g. TM and DVFS). The general approach does not specify the exact form of the taxonomy into which workloads are classified, the monitors and actuators the system need to have, or the design figure of merit. Our examples classify based on differentiating CPU and memory usages and the execution intensiveness, try to maximize IPS/Watt through core-allocation and DVFS, and get information from system performance counters.

A. Workload classification

Real applications do not have precisely tuneable memory usage rates. As a result, information from performance counters is used to derive the classes of all applications running on the system for each control decision cycle. This is based on calculating a number of metrics from performance counter values recorded at set time intervals, and then deriving the classes based on whether these metrics have crossed certain
thresholds. Example metrics and how they are calculated are given in Table III.

### TABLE III: Metrics used to derive classification.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>nipc</td>
<td>(InstRet/Cycles)(1/IPC&lt;sub&gt;max&lt;/sub&gt;)</td>
</tr>
<tr>
<td>ipc</td>
<td>InstRet/ClockRef</td>
</tr>
<tr>
<td>mmipc</td>
<td>(InstRet/Cycles - Mem/Cycles)(1/IPC&lt;sub&gt;max&lt;/sub&gt;)</td>
</tr>
<tr>
<td>cmr</td>
<td>(InstRet - Mem)/InstRet</td>
</tr>
<tr>
<td>urr</td>
<td>Cycles/ClockRef</td>
</tr>
</tbody>
</table>

**Normalized instructions per clock** (nipc) measures how intensive the computation is. It is the instructions per unhalted cycle (IPC) of a core, normalized by the maximum IPC (IPC<sub>max</sub>). IPC<sub>max</sub> can be obtained from manufacturer literature.

**Cycles** is the unhalted cycles counted. Normalization allows nipc to be used independent of core types and architectures.

**Instructions per reference clock** (ipc) contributes to determining how active the computation is. **ClockRef** is the total number of clock cycles given by ClockRef = Freq/Time with Freq and Time from the system software.

**Normalized non-memory IPC** (mmipc) discounts memory accesses from nipc, indicating CPU activity. From experiments with our synthetic benchmark, this shows an inverse correlation to the memory use rate.

**CPU to memory ratio** (cmr) relatively compares CPU to memory activities.

**Unhalted clock to reference clock ratio** (urr) determines how active an application is.

The general relationship between these metrics and the application (workload) classes are clear, e.g. the higher mmipc is, the more CPU-intensive a workload will be. A workload can be classified by comparing the values of metrics to thresholds. Decision-making may not require all metrics. The choice of metrics and thresholds and be made by analysing characterization experiment results. From analysing the relationship between M and the list of metrics from psync experiments, we find that mmipc shows the best spread of values with regard to corresponding to different values of M. This leads to more straightforward arrangements of threshold values between different application classes. Referring to the declared classes in PARSEC applications (ferret is claimed to be CPU-intensive, for instance [23]), this hypothesis is confirmed. As a result, we choose mmipc to differentiate CPU and memory usage rates and urr for differentiating low and high activity. Then thresholds (Table IV) are determined based on our psync characterization database. The other metrics may work better on other platforms and are included here as examples of potential candidates depending on how a psync-like characterization program behaves on a platform with regard to the relationships between M values and the metrics.

### B. Control decision making

This section presents an RTM control algorithm that uses application classes to derive its decisions. The behaviour is specified in the form of two tables: a threshold table (Table IV), used for determining application classes, and a decision table (Table III), providing a preferred action model for each application class.

The introduction of new concurrent applications or any other change in the system may cause an application to change its behaviour during its execution. It is therefore important to classify and re-classify regularly. The RTM works in a dedicated thread, which performs classification and decision making action every given timeframe. The list of actions performed every RTM cycle is shown in Algorithm 1.

### TABLE IV: Classification details.

<table>
<thead>
<tr>
<th>Metric ranges</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>urr of all cores [0, 0.11]</td>
<td>0: low-activity</td>
</tr>
<tr>
<td>mmipc per-core [0.25, 0.35]</td>
<td>1: CPU-intensive</td>
</tr>
<tr>
<td>mmipc per-core [0.25, 0.25]</td>
<td>2: CPU+memory</td>
</tr>
<tr>
<td>mmipc per-core [0, 0.25]</td>
<td>3: memory-intensive</td>
</tr>
</tbody>
</table>

### TABLE V: RTM control decisions.

<table>
<thead>
<tr>
<th>Class</th>
<th>Frequency</th>
<th>A7</th>
<th>A15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>min</td>
<td>single</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>max</td>
<td>none</td>
<td>max</td>
</tr>
<tr>
<td>2</td>
<td>min</td>
<td>max</td>
<td>max</td>
</tr>
<tr>
<td>3</td>
<td>max</td>
<td>max</td>
<td>none</td>
</tr>
<tr>
<td>unclassified</td>
<td>min</td>
<td>single</td>
<td>none</td>
</tr>
</tbody>
</table>

In Algorithm 1 $T_{control}$ is the time between two RTM control cycles. The RTM determines the TM and DVFS of power domains once each control cycle, and these decisions keep constant before the next control cycle. The data from the system monitors (performance counters and power meters) is collected asynchronously. Every core has a dedicated monitor thread, which spends most of its time in a sleep state and wakes every $T_{control}$ to read the performance counter registers. The readings are saved in the RTM memory. This means that the RTM always has the latest data, which is at most $T_{control}$ old. This is mainly done because ARM performance counter registers can be accessed only from code on the same CPU core. In this case, asynchronous monitoring has been empirically shown to be more efficient. In our experiments we have chosen $T_{control} = 500\text{ms}$, which has shown a good balance between RT overhead and energy minimization. The time the RTM takes (i.e. RT overhead) is negligible compared to 500ms for the size of our system. This interval can be easily reduced with slightly higher overheads, or increased with less energy efficiency tradeoffs.

The RTM uses monitor data to calculate the classification metrics discussed in Section V. These metrics form a profile for each application, which is compared against the thresholds (Table IV). Each row of the table represents a class of applications and contains a pre-defined value range for each classification metric. Value ranges may be unbounded. A metric $x$ can be constrained to the range $[c, +\infty)$, equivalent to $x \geq c$. An application is considered to belong to a class, if its profile satisfies every range in a row. If an application does not satisfy any class, it is marked as unclassified and gets a special action from the decision table. An application is also
Algorithm 1 Inside the RTM cycle.

1 Collect monitor data.
2 For each application:
   2.1 Compute classification metrics (Table III).
   2.2 Use metric and threshold table to determine app class (Table IV).
   2.3 Use decision table to find core allocation and frequency preferences (Table V).
3 Distribute the resources between the apps according to the preferences.
4 Wait for $T_{\text{control}}$.

Algorithm 2 Core allocation (TM)

1 For each application:
   1.1 If new app: run on a single A15 and classify if C7 is always reserved for this, but use a lower core (e.g. C4) when possible
   1.2 If current app: classify on its current running core(s);
   1.3 Calculate allocation preference from Table V;
2 For each core type:
   2.1 Give each app needing single cores 1 core
   2.2 Distribute the rest of the cores evenly between apps needing max cores.

unclassified when it first joins the execution. In that case it goes to an A15 core for classification.

The decision table (Table V) contains the following preferences for each application class, related to system actuators (DVFS and core allocation decisions): number of A7 cores, number of A15 cores, and clock frequencies. Number of cores can take one of the following values: none, single, or maximum. Frequency preference can be minimum or maximum. The CPU-intensive application class (Class 1) runs on the maximum number of available A15 cores at the maximum frequency as this has shown to give the best energy efficiency (in terms of power normalized performance) in our previous observations [6].

Table IV and Table V are constructed on this work based on large amounts of experimental data, with those involving PARSEC playing only a supporting role. For instance, although ferret is regarded as CPU-intensive, it is so only on average and has non CPU-intensive phases (see Section VI.A). Therefore Table V is obtained mainly from analysing experimental results from our synthetic benchmark psync (which has no phases), with PARSEC only used for checking if there are gross disagreements (none was found). Because of the empirical nature of the process, true optimality is not claimed.

In this work, we assume that there are always more cores than running applications, without losing generality. The RTM attempts to satisfy the preferences of all running applications. In the case of conflicts between frequency preferences, the priority is given to the maximum frequency. When multiple applications request cores of the same type, the RTM distributes all available cores of that type as fairly as possible. When these conflicting applications are of different classes, each application is guaranteed at least a single core. Core allocation (TM) is done through the following algorithm.

C. RTM governor design

The governor implementation is described in Figure 3, which refines Figure 2. At time $t_i$, application $i$ is added to the execution via the system function execvp(). The RTM makes TM and DVFS decisions based on metric classification results, which depends on hardware performance counters and power monitors to directly and indirectly collect all the information needed. This helps avoid instrumenting applications and/or special API’s (unlike e.g. [4]), providing wider support for existing applications. The TM actuation is carried out indirectly via system functions. For instance, core pinning is done using sched_setaffinity(pid), where pid is the process ID of an application. DVFS is actuated through the userspace governor as part of cpufreq utilities.

VI. EXPERIMENTAL RESULTS

Extensive experiments have been carried out with a large number of application scenarios running on the XU3 platform. These experiments include running single applications on their own and a number of concurrent applications. In the concurrent scenarios, multiple copies of the same application and single copies of different applications of the same class and different classes have all been tested.

A. A Case Study of Concurrent Applications

An example execution trace with three applications is shown in Figure 4. Parts at the beginning and end of the run contain single and dual application scenarios. The horizontal axis is time, and the vertical axis denotes TM and DVFS decisions. Cores C0-C3 are A7 cores and C4-C7 are A15 cores. The figure shows application classes and the core(s) on which they run at any time. This is described by numbers, for instance, 2/3 on core C1 means that App 2 is classified as of Class 3 and runs on C1 for a particular time window. 1/u means that App 1 is being classified. In this example trace, App 1 is ferret, App 2 is fluidanimate, and App 3 is square root calculation.

![Fig. 3: Governor Implementation based on RTM.](image-url)
As can be seen in this concurrent execution scenario, all three workloads, including the conventional Linux CPU-stress application, square root calculation, exhibit multi-class phase behaviour.

The lower part of the figure shows the corresponding power and IPS traces. Both parameters are clearly dominated by the A15 cores.

As can be seen in Figure 4, initial classifications are carried out on C4, but according to Algorithm 2, when C4-C6 are in application execution, C7 is reserved for this purpose, which is not needed in this trace. The reservation of dedicated cores for initial classification fits well for architectures where the number of cores is so large that we can assume that the number of applications is always smaller than the number of cores. This is not an overly restrictive assumption for modern (e.g. the Odroid XU3) and future systems with continuously increasing numbers of cores.

Re-classification happens for all running applications at every $T_{\text{control}} = 500\text{ms}$ control cycle on their running core(s), according to Algorithms 1 and 2. Figure 4 shows the motivation for this re-classification. The same application can have memory usage phases and belong to different classes at different times. This means that OL classification methods, which give each application an invariable class, is unsuitable for efficient energy minimization.

**B. RTM stability and robustness**

Figure 5 shows example traces of the PARSEC apps ferret and fluidanimate being classified whilst running as single applications. It can be seen that the same application can have different CPU/memory behaviours and get classified into different classes. This is not surprising as the same application can have CPU-intensive phases when it does not access memory and memory-intensive phases where there is a lot of memory access. In addition, it is also possible for an application to behave as belonging to different classes when mapped to different numbers of cores. The classification can also be influenced by whether an application is running alone or running in parallel with other applications, if we compare Figure 4 and Figure 5. These are all strong motivations for RT re-classification. The result of classification affects an applications IPS (see Figure 4) and power (see Figure 5).

Algorithm 1 can oscillate between two different sets of classification and control decisions in alternating cycles. This may indicate the loss of stability. The reasons for such oscillations have been isolated into the following cases:

- The control cycle length coincides with an applications CPU and memory phase changes.
- An applications behaviour takes it close to particular threshold values, and different instances of evaluation put it on different sides of the thresholds.
- An application is not very parallelizable. When it is classified on a single core, it behaves as CPU-intensive, but when it is classified on multiple cores, it behaves as low-activity. This causes it to oscillate between Class 0 and Class 1 in alternating cycles.

We address these issues as follows. Case 1 rarely happens and when it happens it disappears quickly, because of the very low probability of an applications phase cycles holding constant and coinciding with the control cycle length. This can be addressed, in the rare case when it is necessary, by tuning the control cycle length slightly if oscillations persist.

Case 2 also happens rarely. In general, increasing the number of classes and reducing the distances between control decisions of adjacent classes reduce the RTMs sensitivity to threshold accuracy, hence Case 2 robustness does not have to be a problem, and thresholds (Table V) and decisions (Table VI) can be tuned both OL and during RT.

Case 3 is by far the most common. It is dealt with through adaptation. This type of oscillation is very easy to detect. We put in an extra class, low-parallelizability, and give it a single big core. This class can only be found after two control cycle s, different from the other classes, but this effectively eliminates Case 3 oscillations.

**C. Comparative evaluation of the RTM**

Complexity: Our RTM has a complexity of $O(N_{\text{app}} \cdot N_{\text{class}} + N_{\text{core}})$, where $N_{\text{app}}$ is the number of applications running, $N_{\text{class}}$ is the number of classes in the taxonomy, and $N_{\text{core}}$ is the number of cores. $N_{\text{class}}$ is...
usually a constant of small value, which can be used to trade robustness and quality with cost. The RTMs computation complexity is therefore linear to the number of applications running and the number of cores. In addition, the basic algorithm itself is low-cost lookup-table approach with the table sizes linear to $N_{\text{class}}$.

Schemes found in existing work, with e.g. model-based [9], machine-learning [21], linear programming [12], or regression techniques [13] [9], have a decision state space size of $((N_{\text{ADVFS}} \cdot N_{\text{ADVFS}}} \cdot (N_{\text{AT}} \cdot N_{\text{AI}})^{N_{\text{app}}})$, where $N_{\text{AT}}$ and $N_{\text{AI}}$ are the numbers of A7 and A15 cores and $N_{\text{ADVFS}}$ and $N_{\text{ADVFS}}^\prime$ are the numbers of DVFS points of the A7 and A15 power domains, for this type of platform. This NP complexity is sensitive to system heterogeneity, unlike our approach.

**Overheads:** We compared the time overheads (OH) of our method with the linear-regression (LR) method found in e.g. [13] and [9]. For each 500ms control cycle, our RTM, running at 200MHz, requires 10ms to complete the trace in Figure 4. Over 90% of this time is spent on monitor information gathering. In comparison, LR requires 100ms to complete the same actions. It needs a much larger set of monitors. The computation, also much more complex, evenly divides its time in model building and decision making. In addition, a modelling control such as LR requires multiple control intervals to settle and the number of control intervals needed is combinatorial with $N_{\text{AT}}, N_{\text{AI}}, N_{\text{ADVFS}}$ and $N_{\text{ADVFS}}^\prime$.

**Scalability:** Our RTM is scalable to any platform as it is a) agnostic to the number and type of application running in concurrently, and b) independent of the number or type of cores in the platform, and their power domains. This is because the complexity of the RTM only grows linearly with increased number of concurrent applications and cores.

**Performance:** Direct comparison is possible only with [9], which studies the same set of benchmarks running on the same platform. As shown in Table VI, which does not take the OH into account for [9], our RTM compares favourably in terms of overall advantages over the Linux ondemand governor. These selected experiments cover single applications and various combinations of applications of different classes running concurrently.

### TABLE VI: PERCENTAGE IPS/WATT IMPROVEMENTS OF THE RTM OVER THE LINUX ONDEMAND GOVERNOR

<table>
<thead>
<tr>
<th>Application scenarios</th>
<th>WLC (w/OH)</th>
<th>LR [9] (w/OH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fluidanimate</td>
<td>124%</td>
<td>27%</td>
</tr>
<tr>
<td>ferret + fluidanimate</td>
<td>68.6%</td>
<td>N/A</td>
</tr>
<tr>
<td>ferret + fluidanimate + bodytrack</td>
<td>46.8%</td>
<td>29.3%</td>
</tr>
<tr>
<td>fluidanimate × 2</td>
<td>24.5%</td>
<td>N/A</td>
</tr>
<tr>
<td>fluidanimate × 3</td>
<td>44.4%</td>
<td>36.4%</td>
</tr>
<tr>
<td>ferret × 2</td>
<td>31.0%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**VII. CONCLUSIONS AND FUTURE WORK**

A runtime management approach is proposed for multiple concurrent applications of diverse workloads running on heterogeneous multi-core platforms. The approach is demonstrated by a governor aimed at improving system energy efficiency (IPS/Watt). This governor classifies applications according to their CPU and memory signatures and makes decisions on core allocation and DVFS. Due to model-free approach, it leads to low RTM complexity (linear with the number of applications and cores) and cost (lookup tables of limited size). The governor implementation does not require application instrumentation, allowing for easy integration in existing systems. Experiments show the governor providing significant energy efficiency advantage compared to existing approaches. Detection of low-parallelizability improves the stability of the governor.

The approach is general in the sense of being agnostic to metrics, platforms, and workloads. It can be extended to the optimization of other performance metrics and different taxonomies of workload classification so long as the metrics in question are related to the classes of the taxonomies. A key enabler is the capability of finding a characterization program, which supports the tuning of all important classification taxonomy parameters. Such a program can then be used to characterize the system platform and derive parameter thresholds and control actions. In the case of this paper, a characterization program $psync$ that accepts the memory usage factor $M$ as an input and implements its tuning according to the input value is developed for this purpose.

This work opens up opportunities for future RTM research including the runtime tuning of such parameters as classification thresholds, control decisions, and RTM control cycles. Another promising direction is using WLC to reduce the state-space learning-based runtime.

**VIII. ACKNOWLEDGMENT**

This work is supported by the EPSRC (project PRiME, grant EP/K034448/1). Aalsaud is also supported by studentship funding from the Ministry of Iraqi Higher Education and Scientific Research.

**REFERENCES**


