Power-Normalized Performance Optimization of Concurrent Many-Core Applications

Matthew Travers, Rishad Shafik, Fei Xia
μSystems Research Group
School of EEE, Newcastle University
Newcastle Upon Tyne, England
{m.travers, rishad.shafik, fei.xia}@newcastle.ac.uk

***Abstract*—Modern operating systems, such as Linux, are capable of executing multiple parallel applications concurrently on many-core platforms. Different applications may have different characteristics with regard to how they exercise the computation and memory resources in these platforms. This paper aims to investigate the impact of such differences on the overall energy consumption and performance tradeoffs. To analyze these tradeoffs, three PARSEC benchmark applications are chosen with different characteristics - memory-intensive, CPU-intensive and a mixture of both. These applications are then concurrently executed in various combinations in experiments, which also help establish optimized run-time controls in terms of dynamic voltage/frequency scaling (DVFS) and thread-to-core allocations at run-time. Such controls are based on state-space models derived through linear regression using the feedback from hardware performance counters. Using the benchmark applications, we demonstrate the effectiveness of our proposed method, which shows up to 23% improvement in power normalized performance expressed as the ratio between instructions per second (IPS) and power consumption (Watt).**

Keywords—Energy efficiency, State-space model, Concurrency, Linear regression, Power-Aware, Many-Core

I. INTRODUCTION

Contemporary computing systems, including embedded and high performance, are exhibiting increased complexities in two dimensions. In one dimension, the number and type of computing resources (cores) are increasing in hardware platforms, and in the other, an increasing diversity of applications are being executed concurrently on these platforms [1] [2]. Managing hardware resources, to achieve energy efficiency, under different application scenarios (single or concurrent) is proving highly challenging due to run-time state-space expansion [5].

To provide control over power and performance tradeoffs, DVFS has been integrated into contemporary processors, e.g. all current Intel and ARM processors [3][6]. In recent years, CPU clock frequencies have increased rapidly to satisfy the increasing performance needs [3][4]. In order to achieve these higher clock speeds the supply voltage has not decreased with the technology feature size. This results in power consumption not scaling with technology feature size reduction. Consequently, power and energy consumption has become a limiting factor to continued technology scaling and performance improvements [7]. DVFS dynamically scales voltage and frequency across a number of pre-set operating points. These have different performance and power consumption characteristics and their choice can be according to workload requirements. DVFS may be controlled at system software level. For instance, it is controlled in Linux with power governors [8], such as `ondemand`, `performance`, `conservative`, `powersave` and `powersave`. These governors use DVFS control 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, primarily because they select only either the maximum or minimum frequency depending on whether the workload is higher or lower than a given threshold [8]. This coarse-grain approach, although serviceable, is not capable of taking advantage of the different degrees of parallelizability of individual applications and producing the most efficient scheduling.

For parallel applications, a useful technique to manage system performance and power consumption is the allocation of cores to execute specific threads. In addition to DVFS states, the number of cores being active and executing influences both performance and power consumption. Core allocations to threads are handled by system software, for example Linux's scheduler [9]. The scheduler seeks to spread the workload of all applications running across multiple cores to achieve maximum utilization. This approach is functional but leaves a number of rooms for improvement. For instance, there is no discrimination about the type of task or thread being scheduled [9], such as CPU-intensive or memory-intensive. Different types of threads require different considerations for performance and power optimizations. Not taking this into account results in indiscriminate sub-optimization.

Over the years substantial research has been carried out addressing run-time energy minimization and/or performance improvement approaches. These approaches have considered a single-metric based optimization: primarily performance-constrained power minimization, or performance improvement within a power budget [16]. For example, Shafik *et al.* proposed a run-time DVFS control approach for power minimization of multiprocessor embedded systems [10]. Their approach uses performance

and user experience constraints to derive the lowest possible operating voltage/frequency points through reinforcement learning and transfer principles. Das *et al.* presented another power minimization approach that models run-time workload characterization to continually update the DVFS and core allocations through multinomial logic regression based predictive controls [11]. A run-time classification of workloads and corresponding DVFS controls based on similar principles is proposed by Wang and Pedram for performance-constrained power minimization [13]. As far as performance optimization within a power budget is concerned, Chen and Marculescu proposed a distributed reinforcement learning algorithm to model power and performance tradeoffs during run-time [12]. Using this model the DVFS and core allocations are adapted dynamically using feedback from the performance counters. Another power-limited performance optimization approach is presented by Cochran *et al.* showing programming model based power budget annotations and corresponding controls [14].

These existing approaches, however, have the following two major limitations. Firstly, these approaches leave further rooms for improvement in terms of energy-efficiency for applications that do not have hard deadlines or performance constraints. For these applications, a single metric (power) based optimization approaches [10][11][13] are likely to favor the solutions that produce the lowest power consumption, while meeting the specified performance requirements. However, such solutions do not automatically render an energy-efficient performance point. The power-limited performance optimization approaches [13][17] ensure the optimal solutions for the given power constraints, but these also do not guarantee the globally optimal energy-efficient performance points especially when there is no hard power limit. A second limitation of the existing approaches is that their run-time implementations are typically based on single application scenarios, and do not lend themselves to generalizations, including concurrent scenarios with different types of applications (with the combinations between CPU- and memory-intensive, for instance).

To address the above limitations, in this paper we make the following *contributions*:

- we propose a run-time method for power normalized performance optimization for heterogeneous application scenarios (single or concurrent) using DVFS and thread-to-core allocations at run-time,
- fundamental to our method is learning the power-normalized performance (expressed as the ratio between instructions per second and power consumption: IPS/Watt) using linear regression based state-space modeling at run-time, and
- we continually adapt the model using the feedback from the performance counters to derive the most energy-efficient performance point for a given application scenario (single or concurrent, CPU- and memory-intensive); to identify the change of application scenarios appropriate run-time thread monitors are established.

To the best of our knowledge, this is the first run-time state-space modeling technique for many-core systems executing concurrent applications, targeting energy-efficient performance optimization.

The rest of the paper is organized as follows. Section II gives a brief overview of the background. Section III explores power-performance tradeoffs for a number of benchmark applications running on the experimental platform. Section IV describes the method of optimizing power-normalized performance. Section V presents experimental results comparing the proposed method with existing Linux governors. Section VI concludes the paper.

II. BACKGROUND

This section discusses CPU power consumption theory, standard Linux power governors, and the experimental platform.

A. CPU power consumption

The power consumed by a processor is related to its operating frequency and voltage. It also has to do with how much work the processor is doing. We call the workload the activity factor. This is how many times on average the transistors switch during every clock cycle. This is important as energy is consumed when transistors switch between states, if a transistor does not switch it only consumes static power (because of the inevitable leakage etc.). The power consumed by a processor can be split up into two parts, static and dynamic power as seen in (1).

$$P_{total} = P_{static} + P_{dynamic} \quad (1)$$

Static power is the power consumed when there is no switching. It is the theoretical minimum amount of power the CPU can consume when turned on. Static power has increased due to transistor sizes getting smaller and transistor density increasing. This creates a higher leakage current and therefore more leakage power [18]. Static power follows complex formulas but for simplicity it has been generally found to be acceptable to conveniently assume that static power is related to voltage linearly [18] [19]:

$$P_{static} = \gamma V + \omega, \quad (2)$$

where V is the supply voltage, and γ and ω are constants.

B. Dynamic Power

Dynamic power is dissipated as a result of switching activities. Switching typically happens when executing tasks, which can include from small OS background tasks to large applications needing to use a lot of processing power. Dynamic power dissipation follows (3) [19]:

$$P_{dynamic} = \alpha C V^2 f, \quad (3)$$

where α is the activity factor (the portion of the circuit that is switching), C is the capacitance of the entire circuit, V is the supply voltage, and f is the frequency. The capacitance is hardware dependent and is a constant for any CPU. The other variables change with the utilization of the CPU, with α being

influenced by thread-to-core allocation and V and f being directly controllable through DVFS.

C. CPU Power Governors

CPU power governors (generally used in Linux operating system) dictates DVFS control reacting to different workloads. The governors can be set using the utility `cpufrequtil`, which reads system files about CPU information and relays it in a user friendly way. For example it can read the files which state the maximum and minimum frequency the CPU can achieve and then choose the minimum frequency when not executing tasks and the maximum frequency when executing tasks, which is what the `ondemand` power governor does [8]. The governors set frequencies directly, and voltage is scaled with the frequency, usually through a number of DVFS points provided by the hardware. In this context, choosing a frequency implies a complete voltage and frequency decision.

Currently there are three major static governors, as follows:

`performance`: a static governor that sets the frequency to the highest possible value,

`powersave`: a static governor that sets the frequency to the lowest possible value,

`userspace`: a static governor that reads information passed by the user to the "scaling_setspeed" file and sticks to that frequency.

There are also two dynamic governors which determine the frequency dynamically according to task states.

`ondemand`: The frequency is scaled up to the maximum value when certain criteria are met and then decreased gradually when other criteria are met [8]. It ramps to maximum frequency whenever there is an increase of activity, and with reductions of activity decrease the frequency in steps. The activity increase and reduction thresholds can be tuned. This governor is best used for user satisfaction.

`conservative`: This works similarly to `ondemand` but instead of jumping to maximum frequency when the up threshold value is met it goes up in steps and then jumps straight to minimum frequency when the down threshold is met. This is best used to save energy whilst still providing some user satisfaction.

Examples of other governors that have been proposed but not widely adopted include [15], which also adapt DVFS dynamically based on the CPU workloads.

D. Experimental platform

All of the experimental data in this paper is obtained from running a Linux platform based on an Intel Core i7 Sandybridge CPU which contains no on-chip GPU facility. This CPU is chosen because it has a reasonable number of hard (4) and soft (8) cores, has no on-chip GPU to distort the power consumption and communications, and as a Sandybridge Core i7 rather than Xeon, has a relatively large number of possible operating frequencies and voltages. The operating system is Ubuntu Linux.

Extra power monitoring facilities are constructed for the experimental platform. This is by inserting a shunt resister into the earth side of the power connection to the CPU. As high-precision current meters tend to have a 1A upper limit, which many CPU operations will exceed, the shunt resister allows the inference of current via measuring voltage.

The performance and power utility Likwid [21] is used to obtain the majority of the experimental data. Likwid makes use of on-chip performance counters (sensors) in Intel CPUs to collect performance and power data. For instance, the Running Average Power Limit (RAPL [22]) counters are accessed to infer power dissipation. Before the main experiments, Likwid was first confirmed to be accurate for the experimental platform through cross-validation with physical power measurements using the shunt resister. The use of performance counters rather than external power measurement in most of the experiments is motivated by the desire of developing a run-time, which for practicality and wide applicability can only rely on built-in sensors (performance counters).

III. Power-Performance Tradeoffs

A number of experiments are designed with standard Linux governors controlling system DVFS during PARSEC benchmark applications. PARSEC benchmarks are suitable for investigating multi-thread operations in either sequential or concurrent executions [20]. These experiments are used to investigate whether different types of applications require different allocations of resources and whether global governors can achieve optimum results.

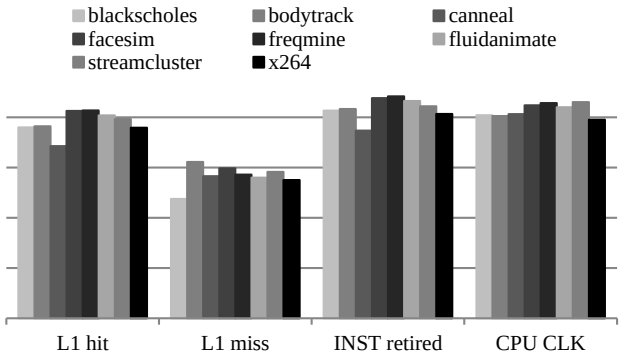
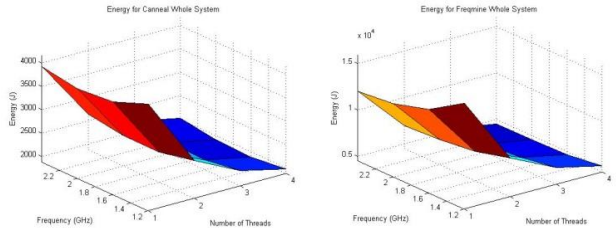


Fig. 1. Performance counter values for a complete run of each application in the PARSEC benchmark suite with `ondemand` governor, recorded by Likwid

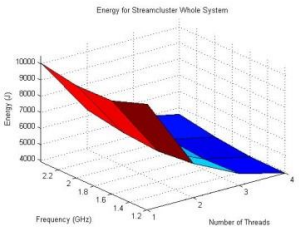
To explore the application state-space, benchmarks of different types are chosen to stress the system in different ways. This profiles how the system reacts to various types of stressing. Three benchmarks are chosen from the PARSEC suite [17]. One is memory-intensive, one is CPU-intensive and one is a mixture of both. Fig. 1 shows performance counter measurements of applications in the PARSEC

benchmark suite, `canneal` (Memory-intensive), `freqmine` (CPU-intensive) and `streamcluster` (Mixture of both) are selected for use in the experiments.

Fig. 2(a-c) shows plots of total energy used (J) to complete a run of each benchmark application with a given frequency (MHz) and number of cores allocated to it, with only one thread allowed per core. Power measurements are in the PKG domain (Cores & Cache), which corresponds with the shunt resister measurement area. Fig. 3(a-c) shows the average instructions per second per watt (IPS/Watt) a benchmark application achieves in one run with a given frequency (MHz) and number of cores allocated to it. A number of observations can then be made.



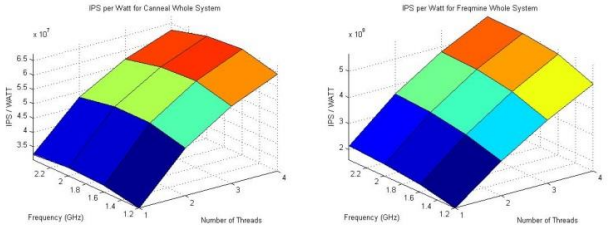
(a) `canneal` (b) `freqmine`



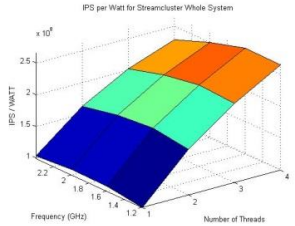
(c) `streamcluster`

Fig. 2. Energy used for a complete run of each application at different operating frequencies and number of cores allocated, data recorded with Likwid

Observation 1: The lowest energy point is always achieved when the application is spread across the maximum number of cores, which in this case is 4. Maximum parallelization reduces the total execution time, and at the same time reduces the number of cores on idle to zero during the run. If energy is only measured during the run, the overall effect is minimizing energy if all cores are busy during the execution. Fig. 2(a-c) shows that the operating frequency needed to achieve minimum energy is application dependent. With reference to Fig. 2(a) and (c) it can be seen that the optimum frequency is at lower frequencies, whereas Fig. 2(b) shows that the optimum frequency is at the maximum frequency of 2.4GHz. This is reasonable as the frequency under study here is the CPU frequency, and the performance of CPU-heavy tasks is more directly related to CPU frequency than that of memory-heavy tasks. The latter typically encounter memory and communication related bottlenecks making them less directly related to CPU frequency – when CPU frequency increases, more clock cycles are needed in waiting for external communications to off-chip memory etc. This heterogeneity of behaviors reinforces the need for a run-time dependent on the type of application running and not for the "one condition suits all" approach currently used by Linux.



(a) `canneal` (b) `freqmine`



(c) `streamcluster`

Fig. 3. Average IPS/Watt for a complete run of each application at different operating frequencies and number of cores allocated, data recorded with Likwid

Observation 2: IPS/Watt closely follows the energy consumption. Fig. 3(a-c) shows that the operating point for maximum IPS/Watt is close to the operating point of minimum energy in Fig. 2(a-c). This is because IPS/Watt is the inverse of energy per instruction. Because each of these benchmarks has a fixed number of total instructions, no matter at what frequency and core state it is run, the measured total energy per benchmark run is proportional to energy per instruction. Experiments are run with Linux governors to assess how well they handle different application types. Table I shows how each of the three benchmarks reacts to different governors. It can be seen that the most energy efficient governor is different for different benchmarks. And it is not always the low power governor (`powersave`). `freqmine` is most energy efficient when under the `ondemand` governor whilst for `canneal` and `streamcluster`, the `powersave` governor is the most efficient.

This indicates that governor performance has poor application independence, and a governor designed to maximize energy savings might not realize lower energy consumption than one that is primarily designed to enhance user experience.

TABLE I. COMPARING GOVERNORS FOR SINGLE APPLICATION EXECUTION

Benchmark	Governors and metrics			
	Governor	*IPS/Watt*	*Energy (J)*	*Total Run-time (S)*
canneal	O[a]	4.61E+07	2746.40	65
	PF[b]	4.60E+07	2754.63	64.6
	PS[c]	6.38E+07	1987.36	125.6
freqmine	O[a]	5.02E+08	5115.81	96.2
	PF[b]	5.00E+08	5138.75	95.4
	PS[c]	4.94E+08	5208.03	291.2
streamcluster	O[a]	1.57E+08	6666.69	131.6
	PF[b]	1.56E+08	6710.81	132.2
	PS[c]	2.65E+08	3948.43	216.2

[a.] ondemand
[b.] performance
[c.] powersave

The impact of these governors is further investigated for concurrent applications of different types. Table II shows the impact of different concurrent application scenarios. The following observations can be made:

Observation 3: IPS/Watt for the system as a whole is the greatest when mixing two CPU-intensive benchmarks (freqmine & freqmine). But the IPS/Watt is also considerably higher when mixing a memory-intensive application with a CPU-intensive one. This would lead to the conclusion that when a memory-intensive application is already running, and both a CPU-intensive and a memory-intensive applications are ready to run, it is better to start the CPU-intensive application for optimum CPU utilization.

Observation 4: Running two copies of the same benchmark produces more or less the same IPS/Watt as running a single copy of that benchmark. Consequently, there is no need to study mixing copies of the same application and in the rest of the paper we will only investigate mixes of different applications.

It has been established that to maximize IPS/Watt, it is necessary to consider the problem on a per-application or at least per-application type basis. In other words, a state in the optimization space needs to include information of application characteristics. This optimization state-space will be defined in the next section, which is dedicated to the development of a run-time control scheme to replace the Linux governors for improving IPS/Watt and related metrics.

IV. POWER-NORMALIZED PERFORMANCE OPTIMIZATION

In this section a run-time control is developed aiming to optimize IPS/Watt, using feedback information obtained from performance counters and current DVFS and core allocation data to calculate outputs to set the system parameters *frequency* and *core allocation*. Core to this method is a model describing the relationship between IPS/Watt and a number of independent variables it depends on. These independent variables describe the states in the optimization state-space.

TABLE II. COMPARING GOVERNORS FOR THE CONCURRENT EXECUTION OF APPLICATIONS

Program	Governors and metrics			
	Governor	*IPS/Watt*	*Energy (J)*	*Total Run-time (S)*
canneal & canneal	O[a]	5.12E+07	4939.33	110
	PF[b]	5.08E+07	4981.45	110
	PS[c]	7.69E+07	3296.11	196.2
freqmine & canneal	O[a]	3.69E+08	7313.76	139
	PF[b]	3.66E+08	7363.25	139.4
	PS[c]	4.09E+08	6592.77	369.2
streamcluster & canneal	O[a]	1.29E+08	9407.77	191.2
	PF[b]	1.29E+08	9410.32	190.8
	PS[c]	2.13E+08	5673.91	318
freqmine & freqmine	O[a]	5.00E+08	10272.94	188.8
	PF[b]	4.99E+08	10310.53	189.4
	PS[c]	4.97E+08	10326.29	575.4
freqmine & streamcluster	O[a]	3.55E+08	8990.58	169.2
	PF[b]	3.44E+08	9440.69	177.8
	PS[c]	3.99E+08	9136.47	503.2
streamcluster & streamcluster	O[a]	1.59E+08	17231.34	342
	PF[b]	1.59E+08	16712.37	332
	PS[c]	2.65E+08	8427.56	464.6

A. IPS/Watt State-Space Model

To establish the model, linear regression is used to derive the relationship between the dependent variable (i.e. IPS/Watt) and the independent predictor variables related to the operating state (e.g. task mapping, VFS, etc.) [23].

Such a relationship is defined by a hypothesis function

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \Theta^T X, \qquad (4)$$

where x_i is a predictor, n is the number of predictors, and θ_i is a fitting coefficient. Linear regression means that the hypothesis function is linear in the fitting coefficients and can be expressed in matrix form as on the right hand side of (4).

The coefficient values in Θ need to be chosen so that some measure of error is minimized. The usual practice, followed in this work, is to minimize the mean-squared prediction error, known as the least squares method. The least squares method is widely implemented in mathematical and engineering tools such as Matlab [24], which is used in this work.

As the physics behind both performance and power is well established, it is possible to decide the identities of the

predictor variables without going through such more complex procedures as principle component analysis [25].

When considering which predictor variables to choose, in addition to the knowledge of how certain independent variables may be related to the dependent variable in the sense of physics, an equally important factor is being able to know the value of each independent variable at run-time without incurring high costs. This is because for a model to be useful, its independent variable values need to be available.

The dependent variable, IPS/Watt consists of two parts, instructions per second and power dissipation. IPS is related to clock frequency f through *Clock Cycles per Instruction (CPI)*:

$$IPS = \frac{fN}{CPI}, \quad (5)$$

where N is the number of cores used for execution. Note that both f and N are operating state variables. Frequency is part of the DVFS state and the number of cores is part of the thread-to-core allocation state. *CPI*'s relationship to the operating state is discussed later.

Power consumption is related to the DVFS state and how many-cores are being used for execution. Even though (3) shows that power is related to both frequency and voltage, the DVFS state is described by a single independent variable as voltage and frequency always come in DVFS pairs in CPUs that support DVFS. Voltage is hence dependent on frequency and is not itself an independent variable. For a modern processor such as the i7 in our experimental platform, voltage relates to frequency largely linearly outside special operating modes such as turbo boost:

$$V = \varphi f + \beta, \quad (6)$$

where φ and β are constants. In turbo boost or overclocking modes, the frequency saturates hence voltage is greater than predicted by (5). However since our experiments only go up to 2.4GHz which is within the normal operation mode, the relationship between voltage and frequency follows (5).

Consequently, from the context of known relations, we only need three predictor variables, *CPI*, *f*, and *N*. The latter two (*f* and *N*) are readily available during run-time as they are set by the run-time control itself. *CPI* however is not as straightforward. It is certainly application dependent as each application includes a unique organization of instructions from the instruction set. In this regard it can be said that *CPI* is a thread-to-core allocation state. However, each application may include different instructions in different branches and therefore its transient *CPI* cannot be determined as it changes throughout the code and may be data dependent. Moreover, for modern systems with speculation and other sophisticated optimization techniques, each individual instruction may require a variable amount of clock cycles depending on the execution context. It may be possible for code to be pre-analysed and instruction data annotated for *CPI* modelling but this is yet far from reality.

On the other hand, *CPI* values can typically be inferred from performance counter data. For Intel CPUs, transient average *CPI* can be directly obtained at run-time through reading two on-chip performance counters, `INST_RETIRED.ANY`, which monitors the number of instructions retired (completely executed), and `CPU_CLOCK_UNHAULTED.CORE`, which counts the number of clock cycles for a core not in a halt (idle) state. In this sense, *CPI* can be regarded as a performance counter feedback state. This is an example of feedback making a run-time possible where offline modelling is impractical.

By combining (2), (3), (5) and (6), a hypothetical model for IPS/Watt can be established over the operating state-space described by the three predictor variables *CPI*, *f*, and *N*:

$$\frac{IPS}{Watt} = \frac{fN}{CPI(\theta_0 + \theta_1 Nf + \theta_2 Nf^2 + \theta_3 Nf^3)}, \quad (7)$$

where θ_i, $i = 0 \sim 3$, are constants combining the effects of all constants in the power model.

The values of these constants can be found through linear regression (4), where they are regarded as members of Θ. The linearity requirement means that the method is used to determine to determine the formula for power before that is combined with the formula for IPS in (7). During the exercise we found that it is possible to discard the f^3 term and simplify the f^2 term by taking the variable N out of it, without affecting the R-squared values (all R-square values are greater than 0.95 after the simplification). This results in a simpler model for use at run-time.

Further model simplification is investigated since we observed that the application dependency of these models, whilst certainly existing, is not very high. It is decided that a single model be constructed through averaging the coefficients from the three application-specific models. This is shown below.

$$\frac{IPS}{Watt} \approx \frac{fN(1 \times 10^9)}{CPI(11.061 + 0.645fN + 1.4351f^2)} \quad (8)$$

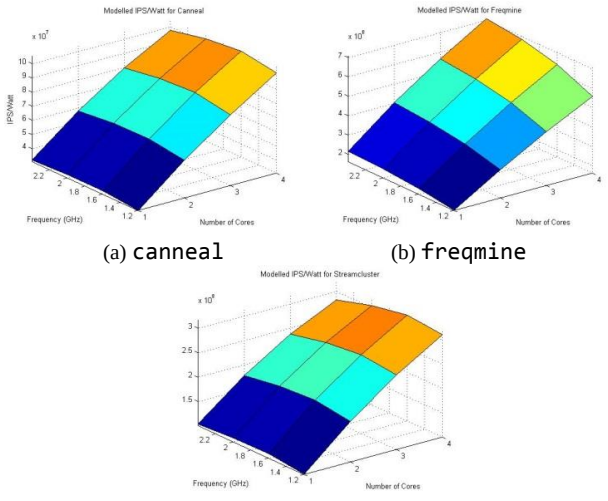
In (8), the frequency is given in GHz and not in Hz.

The applicability of this level of model simplification is based on the assumption that the model will be used in a run-time optimization scheme which has limited scope of tuning its output variables. Instead of continuous changes of *f* and *N*, the run-time can only choose 4 integer *N* values and 7 discrete *f* values. So long as the optimal output states calculated by formula (8) correspond to those observed from experimental data, in the decision space the model's accuracy may be deemed sufficient.

To validate the model, *CPI* information from the data used in making Fig. 3(a-c) is put into (8) and the output plotted. Results of this can be seen in Fig. 4(a-c).

The surfaces in Fig. 4(a-c) strongly resemble those in Fig. 3(a-c). Actual values of IPS/Watt differ slightly from recorded results in Fig. 3(a-c) but the shapes of the surfaces are very similar. More importantly, the operating states for maximum IPS/Watt are the same in both. This simplified application independent model is therefore chosen for use in subsequent sections. Note that the modelling process starts

from predictions according to physics, and ends with a purely pragmatic model for practical use.



(a) `canneal` (b) `freqmine`

(c) `streamcluster`

Fig. 4. Average IPS/Watt outputs from model when input with CPI information from `canneal`, `freqmine` and `streamcluster`

B. *Run-Time Algorithm Design*

The main goal of the run-time optimization is to achieve maximum IPS/Watt of the system as a whole. To achieve this it makes decisions on DVFS and thread-to-core allocation. In this study a relatively coarse grain is chosen and thread is explored at application level. Hence the scheduling is done at the level of application to core allocation. Core allocation in this scheme favors applications with lower *CPI* values. This in principle should lead to higher IPS. Once cores have been allocated the model is used to estimate the IPS/Watt achieved by that core allocation. The frequency will then be stepped up or down depending on previous results and another estimation of IPS/Watt will be performed. These results are compared and a decision about operating frequency for the particular allocation is made.

This run-time optimization requires a computation overhead, and it is important to decide how frequently it should be run. To reduce overhead it should be run as infrequently as possible. To ensure better control response and quality it should be run as frequently as possible. Control quality and response time are related to how frequent the applications change in the execution. For instance, if the control is activated at a frequency lower than the Nyquist frequency of applications themselves it would be completely ineffective, and probably counter-productive [26]. In this study the run-time optimization sampling period is chosen to be 0.5Hz, a value high enough to ensure a reasonably high rate of response – it is much higher than the Nyquist frequency of any of the applications as these run for hundreds of seconds. It is also very low compared with the clock frequencies in the operation space for a very low overhead.

The run-time optimization follows Algorithm 1. In each iteration, it starts by checking if any new application has started. Then it obtains the *CPI* of the new application. This is put into the model to calculate optimal IPS/Watt for potential core allocations to determine core allocation, which is then implemented. Then frequency decisions are made and implemented. Part of the run-time Python script that is used in the experimental studies is shown in Fig. 5.

Algorithm 1: Run-time optimization

1. Check PID changes
2. **If** application scenario changed?
3. Obtain PID of new application
4. Calculate CPI of application
5. Calculate IPS/Watt using model (8)
6. Allocate cores to application
7. Change frequency for max(IPS/Watt)
8. **End if**
9. Wait for next activation

```
while true
do
    if [[ $PID -eq 0 ]]
    then
        governor="ondemand"
        if [[ $m -eq 1 ]]
        then
            set_governor
            m=0
        fi
        PID=$( cat PID.txt )
        echo "no program running"
        sleep 1
    else
        echo "new program added"
        governor="userspace"
        PID=$( cat PID.txt )
        j=0
        PID_CPI_array
        j=0
        if [[ $m -eq 0 ]]
        then
            set_governor
            m=1
        fi
        j=0
        CORE_Allocation
        j=0
        Linear_model
        j=0
        sleep 2
    fi
done
```

Fig. 5. Partial code of the main run-time script to demonstrate Algorithm 1

V. Experimental Results

To validate whether the proposed technique works, a series of experiments are carried out. These experiments are the same as the ones which produced Table I and Table II except the run-time optimization algorithm is used instead of the standard Linux governors.

A. *Single Application*

The first set of experiments aims to show that the run-time can outperform the default governors in energy consumption

and IPS/Watt for applications running alone. Each benchmark is run till completion and the performance counters are measured with Likwid. Table III shows how the run-time script's results compare to the default Linux governors in Table I in terms of percentage increase or decrease, the following observations can be made from these results.

TABLE III. COMPARING PROPOSED METHOD WITH GOVERNORS FOR SINGLE APPLICATION EXECUTION

Program	Governors and metrics			
	Governor	*IPS/Watt*	*Energy (J)*	*Total Run-time (S)*
canneal	**R[d]**	**7.85E+07**	**1949.35**	**104.80**
	O[a]	4.61E+07	2746.40	65
	PF[b]	4.60E+07	2754.63	64.6
	PS[c]	6.38E+07	1987.36	125.6
freqmine	**R[d]**	**5.58E+08**	**4615.36**	**170.00**
	O[a]	5.02E+08	5115.81	96.2
	PF[b]	5.00E+08	5138.75	95.4
	PS[c]	4.94E+08	5208.03	291.2
streamcluster	**R[d]**	**2.63E+08**	**3994.46**	**188.40**
	O[a]	1.57E+08	6666.69	131.6
	PF[b]	1.56E+08	6710.81	132.2
	PS[c]	2.65E+08	3948.43	216.2

[d.] Run-time

Observation 5: When running `canneal` with the run-time script a saving in energy of 1.91% can be achieved compared to the lowest energy achievable with Linux governors. At the same time the time taken to complete the application improves. The total run-time decreases by 16.5% which is a large performance increase for a decrease in energy. The IPS/Watt sees an increase in 23.1% which confirms that the run-time works as planned for `canneal`.

Observation 6: The energy saved with `freqmine` using the run-time script is 9.78% when compared to the lowest energy governor `ondemand`. There is no performance increase with the run-time script when compared to `ondemand` with it taking 76.72% longer. This is not surprising as the computation time is not the optimization target. If we compare it to the governor that is supposed to give the lowest energy which is `powersave`, there is a decrease of 11.38% in energy but also a decrease of 41.26% in the time taken. IPS/Watt sees an increase of 11% when compared to `ondemand` and 12.89% compared to `powersave`.

Observation 7: `streamcluster` is the only application not to see an increase in IPS/Watt and a decrease in the energy used. When the run-time script is compared to the lowest energy Linux governor we see an increase of 1.17% in the energy used. There is also a decrease in IPS/Watt of 0.76%. There is however a performance increase with a decrease of 12.86% in the total time taken to run the application. Reasons for this run-time using slightly more energy than the Linux governors is that the optimum point reached is at the lowest frequency 1.2GHz which is chosen by the `powersave` governor already.

The data collected from the particular example with Stremcluster compared with the proposed run-time, both tending to choose the same execution state, provides an indication of the overhead of the run-time optimization itself. The results show that this overhead is very low (less than 1% in IPS/Watt).

B. Concurrent Applications

More experiments are designed to show how the proposed run-time optimization behaves when controlling different applications running concurrently. Again the results are compared to earlier results obtained by the Linux governors in Table II. Table IV shows these results. Several observations can be made.

TABLE IV. COMPARING PROPOSED METHOD WITH GOVERNORS FOR CONCURRENT EXECUTION OF DIFFERENT APPLICATIONS

Program	Table Column Head			
	Governor	*IPS/Watt*	*Energy (J)*	*Total Run-time (S)*
canneal & freqmine	**R[j]**	**4.43E+08**	**6107.12**	**235.40**
	O[a]	3.69E+08	7313.76	139
	PF[b]	3.66E+08	7363.25	139.4
	PS[c]	4.09E+08	6592.77	369.2
freqmine & streamcluster	**R[j]**	**4.18E+08**	**8773.43**	**328.00**
	O[a]	3.55E+08	8990.58	169.2
	PF[b]	3.44E+08	9440.69	177.8
	PS[c]	3.99E+08	9136.47	503.2
streamcluster & canneal	**R[j]**	**1.96E+08**	**6434.56**	**297.40**
	O[a]	1.29E+08	9407.77	191.2
	PF[b]	1.29E+08	9410.32	190.8
	PS[c]	2.13E+08	5673.91	318

Observation 8: The experiment of `canneal` and `freqmine` running concurrently shows promising results with a reduction in energy consumed by 7.37% and an IPS/Watt increase of 8.10% over the lowest energy governor (`powersave`). Not only is there a reduction in energy there is also a reduction in total run-time of 36.24%. The run-time script works as planned for the mix of these two applications.

Observation 9: The mix of `freqmine` and `streamcluster` also shows a reduction in energy over the lowest energy governor (`ondemand`). A reduction in energy of 2.42% and an increase in IPS/Watt of 17.79% is shown when compared to the `ondemand` governor. We do however see an increase in the time taken of 93.85%. Time taken is not the focus of this

script though and is only a bonus if it completes in less time. If we compare the results of the governor that should be using the least power (`powersave`) we not only see a reduction in energy of 3.97% but also a reduction in total time taken by 34.82%.

Observation 10: Lastly the mix of `canneal` and `streamcluster`. We do not see a reduction in energy here but rather an increase of 13.41%, this is not as intended. Possible reasons for this are down to the fact that `streamcluster` earlier when running on its own had problems decreasing energy. `streamcluster` has a lower CPI value than `canneal` and therefore is assigned more cores than `canneal`. This could lead to the big energy increase as the problematic application is taking up a majority of the processing power. Further experiments and investigations need to be done to assess what is causing the problems with `streamcluster`.

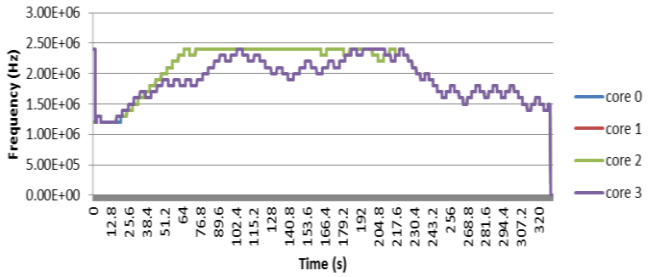


Fig. 6. Frequencies over time for all 4 cores when running `streamcluster` and `freqmine` together

The results from Table IV are promising and shows that a script such as the one outlined in this paper may have real world energy saving benefits. Fig. 6 shows how the frequency is dynamically changed over time and different for each core and hence application. We can see `freqmine` on cores 0/1/2 which rise up to maximum frequency and hold that frequency for the duration of executing `freqmine` with only small deviations from it. Whereas we see `streamcluster` constantly changing its frequency as the application changes the type of processing it is doing (and hence its *CPI*). This is in stark contrast to the other governors where the frequency simply holds at one frequency across all 4 cores for the entire run of the applications.

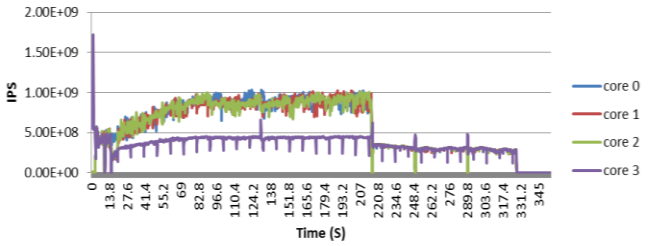


Fig. 7. IPS over time for `freqmine` and `streamcluster` running together

In Fig. 7 we can see from the IPS figures how these two applications are split up between the cores. The application with the higher *CPI* (`streamcluster`) is running on only one core (core 3). `freqmine` with a lower *CPI* is assigned to run on the other 3 cores 0/1/2. This differential allocation allows further IPS/Watt optimizations.

VI. Conclusions

An optimization scheme targeting power-normalized performance has been developed for controlling concurrent application executions on platforms with multiple cores. The method is centered around an operational state-space model, which has independent variables that describe the operational space, i.e. the input and output of the control mechanism. The dependent variable is the metric being optimized. In choosing the power-normalized performance as the metric and investigating the concurrent execution of different applications, this work fills significant gaps in the research literature.

In the first instance, models are obtained off-line from experimental data. Explorations with model simplification are shown to be successful as by and large optimal results are obtained from using these models in a run-time control algorithm compared with existing Linux governors. In many cases the improvements obtained are quite significant.

The method with which the state-space model which underpins this method is obtained, linear regression with least squares approximation, can be used in online learning based solutions. This opens up future research possibilities where the model is tunable during run-time for better optimization results and remove the need for using the same average model for different applications. Another future research potential opened up by this work is in the investigation of other, more sophisticated optimization algorithms. The experimental platform constructed during this work will facilitate these kinds of research in the future.

For experimental purposes, our method was implemented on an Intel Core i7 platform with 4 cores, running Linux system software as a case study. However, the underlying modelling and run-time methodology can be applied to any platform, whilst the implementation of which will be platform-specific. A hard prerequisite for the platform is that some form of feedback mechanism is available to avoid the need for obtaining high-precision off line models. Performance counters have become standard on contemporary platforms to make this a non-issue. The authors do have access to experimental platforms with modern heterogeneous mobile processors and high performance computing platforms with a large number of cores. It is planned that work will immediately start on extending this research over those kinds of platforms. It is expected that modern embedded and high performance systems will benefit from using the proposed method to achieve energy efficiency.

Acknowlegements

This work is part of the EPSRC PRiME project (EP/K034448/1). The authors wish to thank members of the PRiME team in Newcastle University for fruitful discussions.

REFERENCES

[1] A. Prakash, S. Wang, A. E. Irimiea and T. Mitra, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, New York, NY, 2015, pp. 208-215.

[2] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty and A. Herkersdorf, "High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations," *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, Madrid, 2010, pp. 229-234.

[3] Intel Corporation, "Timeline of Processors," Intel, 2012. [Online] http://www.intel.co.uk/content/www/uk/en/history/history-intel-chips-timeline-poster.html. [Accessed 17 December 2015].

[4] Intel Corporation, "Intel Core i7-6700k Processor," Intel, 2015. [Online]: http://ark.intel.com/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz. [Accessed 17 December 2015].

[5] A.-C. Orgerie, M. D. de Assuncao, L. Lefevre. 2014. A survey on techniques for improving the energy efficiency of large-scale distributed systems. ACM Comput. Surv. 46, 4, Article 47 (March 2014), 31 pages.

[6] S. Mittal, "A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems," Int. J. of CAE&T, 2014.

[7] S. Borkar, "Design challenges of technology scaling," in *IEEE Micro*, vol. 19, no. 4, pp. 23-29, Jul-Aug 1999.

[8] V. Pallipadi and A. Starikovskiy, "The Ondemand Governor," Intel Open Source Technology Center, Ottawa, 2006.

[9] L. A. Torrey, J. Coleman and B. P. Miller, "Comparing Interactive Scheduling in Linux," University of Wisconsin, Madison, http://pages.cs.wisc.edu/~ltorrey/papers/torrey_spe06.pdf.

[10] R. Shafik; S. Yang; A. Das; L. Maeda-Nunez; G. Merrett; B. Al-Hashimi, "Learning Transfer-based Adaptive Energy Minimization in Embedded Systems," in IEEE Trans on CAD of IC&S, vol. PP., no. 99, pp.1-14.

[11] A. Das, R. Shafik, G.Merrett, B. Al-Hashimi, A. Kumar, B. Veeravalli "Multinomial logistic regression-based workload noise classification and adaptive frequency scaling for energy minimization of embedded systems," DATE'15, Grenoble, FR, 19 - 21 Mar 2015.

[12] Z. Chen, D. Marculescu. "Distributed reinforcement learning for power limited many-core system performance optimization," DATE '15. Grenoble, FR, pp.1521-1526.

[13] Y. Wang; M. Pedram, "Model-Free Reinforcement Learning and Bayesian Classification in System-Level Power Management," in IEEE Transactions on Computers , vol.PP, no.99, pp.1-1.

[14] R. Cochran, C. Hankendi, A. Coskun, S. Reda, "Pack & Cap: adaptive DVFS and thread packing under power caps." MICRO-44. ACM, New York, NY, USA, 175-185.

[15] V. Spiliopoulos, S. Kaxiras, G. Keramidas, "Green governors: A framework for Continuously Adaptive DVFS," in International Green Computing Conference and Workshops (IGCC), 2011, pp.1-8, 25-28 July 2011.

[16] C. Hankendi, and A. K. Coskun. "Adaptive power and resource management techniques for multi-threaded workloads." In IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 2302-2305, 2013.

[17] C. Bienia, S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Charaterization and Architectual Implications," PACT 2008. ACM, pp. 72-81, New York, NY, USA, 2008.

[18] N. S. Kim *et al*., "Leakage current: Moore's law meets static power," in *Computer*, vol. 36, no. 12, pp. 68-75, Dec. 2003.

[19] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," in *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498-523, Apr 1995.

[20] C. Bienia, S. Kumar and Kai Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," *IEEE International Symposium on Workload Characterization, 2008*, Seattle, WA, 2008, pp. 47-56.

[21] likwid - light weight performance tools, [Online]: https://github.com/RRZE-HPC/likwid/wiki.

[22] M. Hähnel, B. Döbel, M. Völp, H. Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (January 2012), 13-17.

[23] J. Cohen et al. Applied Multiple Regression/Correlation Analysis For The Behavioral Sciences. Routledge, 2013.

[24] Matlab, http://uk.mathworks.com/products/matlab/.

[25] B. Moore, "Principal component analysis in linear systems: Controllability, observability, and model reduction," in IEEE Trans on Automatic Control, vol. 26, no. 1, pp. 17-32, Feb 1981.

[26] R. Poley, Control Theory Fundamentals, CreateSpace Independent Publishing Platform; 3 edition (24 Mar. 2015).