RePP-C: Runtime Estimation of Performance-Power with Workload Consolidation in CMPs

Rajiv Nishtala, Xavier Martorell
Universitat Politècnica de Catalunya and Barcelona Supercomputing Center
Barcelona, Spain
Email: firstname.lastname@bsc.es

***Abstract*—Configuration of hardware knobs in multicore environments for meeting performance-power demands constitutes a desirable feature in modern data centers. At the same time, high energy efficiency (performance per watt) requires optimal thread-to-core assignment. In this paper, we present the runtime estimator (*RePP-C*) for performance-power, characterized by processor frequency states (P-states), a wide range of sleep intervals (Cl-states) and workload consolidation. We also present a schema for frequency and contention-aware thread-to-core assignment (*FACTS*) which considers various thread demands.**

The proposed solution (*RePP-C*) selects a given hardware configuration for each active core to ensure that the *performance-power* demands are satisfied while using the scheduling schema (*FACTS*) for mapping threads-to-cores. Our results show that *FACTS* improves over other state-of-the-art schedulers like Distributed Intensity Online (*DIO*) and native Linux scheduler by 8.25% and 37.56% in performance, with simultaneous improvement in energy efficiency by 6.2% and 14.17%, respectively. Moreover, we prove the usability of *RePP-C* by predicting performance and power for 7 different types of workloads and 10 different QoS targets. The results show an average error of 7.55% and 8.96% (with 95% confidence interval) when predicting energy and performance respectively.

I. Introduction

Managing energy efficiency (performance per watt) requirements has become a key issue in all types of computing systems, ranging from hand-held devices to modern data centers. The energy efficiency requirements can be satisfied at a very fine granularity to the level of workloads or cores in multi-core environments. On the level of data centers, mismatch in Quality of Service (QoS) requirements is usually observed, this leads to violation of the performance or power QoS guarantees for a given workload at a given time instance. Satisfying these QoS requirements needs data center administrators to control performance or power at runtime for a given workload or a node.

This is an online optimization problem that may be represented in various forms: for example minimizing energy consumption subject to a performance target and peak power constraint, or maximizing performance subject to a peak power constraint or to enable energy-efficient cluster management in data centers (which thread is assigned to each core and what frequency to set each core when) and typically use the native hardware (HW) capabilities. Irrespective of the precise formulation, any solution, whether optimal or heuristic-based, requires a fast and accurate model to predict how a potential change in P-State (DVFS, Dynamic Voltage/Frequency Scaling), C-State or other low-level power state would translate into real thread performance and power demand.

Traditionally, power and performance controlling algorithms are iterative P-State based. These algorithms are based either on the thread behavior history [1] or in a given time quantum and set the P-States setting for the next quantum. However, these approaches require multiple iterations before power-performance criteria are satisfied and this leads to massive violations in power-performance budgets for data centers running on strict service level agreements.

In lieu of quick, yet non-reactive approaches, prediction based algorithms provide the benefit of quick response and reaction for a small fraction of computation costs. Since threads have a broad spectrum of computation and memory requirements among others, the hardware performance monitoring counters (PMCs) available on most commercial processors provide users the means to understand the thread behavior. Using such counters, predictive algorithms determine if the thread is scalable at runtime and select the most appropriate configuration based on online optimization problem in a single time instance.

Modern architectures with performance-power control mechanisms such as P-States and idle-cycles (Cl-States) [4], do not allow users to satisfy QoS guarantees below a certain threshold with all cores active [23] (any core with a thread running is defined as an active core). Our insight is that using both prediction techniques and scheduling threads [3], [5]–[8], [13], [18], [19] to avoid negative interference, can minimize power consumption.

The main contribution of this paper include:

(1) A frequency and contention-aware thread co-running schema (**FACTS**) for workload balancing with core consolidation, where each core operates at a different core configurations. We evaluate and compare the energy efficiency and performance over other state-of-the-art schedulers like native Linux scheduler and Distributed Intensity Online (**DIO**) [3] while considering various workloads from the SPECcpu2006 benchmark suite [2]. In particular, the algorithm developed improves in energy efficiency over DIO by 6.2% mean (8.25% mean in performance) and the native Linux scheduler by 14.17% mean (37.56% mean in performance).

(2) Nishtala et al [4] devised a technique to predict power and performance at runtime (RePP) for a multicore Intel architecture using PMCs. We extend [4] for runtime estimation

978-1-5090-5117-5/16/$31.00

of performance-power with workload consolidation (**RePP-C**). We show that RePP-C scales well and is effective to multicore architectures that require meeting QoS guarantees. RePP-C provides a mapping from processor performance states (P-States), Cl-States and core consolidation to QoS like performance (Millions of Instructions Per Second (MIPS)) and power (in W), respectively. We observe average errors of 7.55% and 8.96% (with 95% confidence interval) when predicting energy and performance, respectively.

II. Related Work

Prior research focused on exploring the energy efficiency or performance benefits in multicore processors, with emphasis on thread-to-core mappings to avoid negative interference due to shared resource contention [3], [5]–[8], [13], [18], [19]. The existing literature also refers to guaranteeing a certain performance or power consumed using prediction techniques.

[Performance-Power prediction]: In particular, Bellosa [9] used PMCs at run time to build a power-aware policy at OS level. Isci first showed that using PMCs it is possible to detect fine-grained thread phases [10] and then show breakdown of power per component using multilinear models [11].

In contrast to Mccullough et al. [12], who propose that linear regression of power tends only to work in restricted scenarios and will tend to over-fit based on thread type. Our results and prior work [11], [13]–[16] show that linear regression models built using a small training dataset do predict power and performance at runtime for a broad range of benchmarks, which are not a part of the training dataset, with high accuracy. S. Srinivasan et al. [17] predict the performance of threads running on heterogeneous cores, that is from one core type to another, using closed expressions. These expressions, however, do not suffice for a generic approach.

Pack & Cap [18] predicts performance with an offline analysis trained using multi-nomial logistic regression classifier. When a change in configuration is required at runtime, this classifier returns the best candidate operating configuration. In contrast, RePP-C builds power and performance models using PMCs, giving a more generic approach with low complexity. Since, RePP-C can predict performance-power for a distinct combination of P-States, Cl-States and core consolidatoin simultaneously. This makes RePP-C a good black-box for a single step fine grained per-core power or performance online optimization problem solver without external power meters or using thread signatures [19].

The recent work on PEPP [16] (Performance, Energy and Power Predictor) proposes a system-level performance [20] and power model by taking advantage of the hardware PMCs available in recent *commodity* AMD processors to estimate the total number of leading loads, and in turn, predicts performance and power across P-States states. In contrast to PEPP, RePP-C improves in three distinct ways: (1) RePP-C builds performance and power models based on basic PMCs available across all architectures (Intel, AMD and ARM). However, the leading loads counter used in PEPP is exclusive to AMD processors [21]. (2) In addition to providing system level performance and power management techniques, RePP-C also facilitates thread level power and performance control. This is especially useful in multi-node, multicore data centers consisting of numerous threads having different performance and power requirements. (3) Finally, RePP-C provides fine grained, in terms of time and precision, power and performance control because it also consider Cl-State.

[Contention-aware scheduling]: DIO [3] is a thread scheduling technique directed towards reducing the contention for shared resources such as cache and memory. DIO dynamically schedules threads by spreading contention equally among different memory domains, taking into consideration the cache intensity of the threads. A memory domain is defined as a set of cores sharing a last level cache (LLC).

III. FACTS: Frequency and Contention aware thread scheduling

The most important research challenges when addressing workload balancing and co-running threads at runtime is the need to select the appropriate co-runner that will minimize the negative impact on performance or power. Scheduling algorithms have been proposed for co-running threads based on inter-core [5], [6] and intra-core [7] contention.

We present our solution that addresses the challenges in inter-core contention to improve distribution of memory bandwidth across all cores while trying to maximize energy efficiency with heterogeneous P-States. To cope with the changes in the thread behavior, mapping of threads to core(s) is done periodically (*reassignment interval*). The mapping of threads is determined by taking into account the memory bandwidth requirements and the current P-State of the core.

Specifically, we aim to minimize the difference in the total number of LLC misses per second in a given memory domain. To distribute the contention across memory domains, we assign the thread with the least number of cache misses (high MIPS and low stall cycles, *destination thread*) and highest number of cache misses (low MIPS and high stall cycles, *target thread*) to the core with the highest operating P-State. Similar to operating in heterogeneous multicore platforms, where threads with high IPS and low LLC, are mapped to big cores to maximize energy efficiency.

[Algorithm description]: FACTS is run periodically with a period determined by the reassignment interval of 1 second (empirically determined). The initial thread mappings are assigned by the Linux scheduler, which allocates threads to cores to balance the workload among all available cores in the system [22]. After a reassignment interval, we read the P-State of the each core (the core P-State is fixed, refer Section V-A) and gather the LLC misses using PMCs to determine the memory requirements of the thread. Thereafter, sort the threads in the descending order of LLC misses and sort active core based on the descending order of current P-State of the cores. Finally, FACTS maps threads to cores at runtime (low cache misses to cores with higher P-State). Thereafter, the remaining threads, if any, are mapped to the cores in reverse order to the cores, that is, high cache misses to the core with high

P-State and low cache misses. This mapping given by FACTS helps spread the total contention uniformly among all cores.

IV. RePP-C: Runtime Estimator for Performance Power with Core Consolidation

This section proposes RePP-C (Runtime Estimator for Performance-Power with Core Consolidation) to model and estimate performance-power in multicore systems with core consolidation. We begin the section with an overview of RePP, the way to select a configuration in RePP-C to meet QoS guarantees and then we define QoS. Then show the procedure to build the model to predict performance-power using multilinear regression models with core-consolidation using FACTS and RePP-C. Finally, we describe a technique to select the configuration.

[Overview of RePP]: RePP is a scalable modeling and prediction technique on multicores that works for single-threaded and multiprogrammed workloads alike. It is aimed at controlling performance and power in data centers that execute hundreds of new instances of threads.

RePP does not leverage thread signatures or similarities between threads to predict the performance and power. Instead it is based on online monitoring of basic PMCs, which provide thread behavior at runtime. Such data is then fed into statistical tools to predict performance and power, making RePP fast, accurate, and architecture-agnostic.

RePP is divided into two phases: offline and online. In the **offline phase**, we build single core models by extensive profiling of a small set of benchmarks at various P-States and Cl-States to build multilinear regression models. We build one multilinear regression model per P-State. We show that the basic version of RePP provides accurate predictions for single and multicore architectures with low error and marginal overheads [4]. In the **online phase**, we use the multilinear regression models to predict, at all P-States and Cl-States, and control performance and power per core in the multicore environment. To obtain the total performance or power in multicore architectures, we aggregate the results from each of the single cores to estimate the system performance and power consumption (see [4]).

In the offline phase, we profile the activity in the microarchitectural components Front End (FE), Integer units (INT), Floating Point units (FP) branch predictor unit (BPU), private L1 (L1 cache), private L2 (L2 cache), LLC (L3 cache) and memory subsystem (MEM). Since, these components do have PMCs to record their activity directly, the **activity ratio (AC)** of these components is computed using PMCs activity formulas. The activity ratio is defined as the components average number of uops per cycle (uops/cycle). The complete set of activity formulas for can be seen in [4].

[RePP-C configuration selector]: To enforce strict QoS guarantees, RePP-C controls the core's P-State, Cl-State of the next interval. We select a configuration per core every 250ms by doing a linear search for the P-State which has the nearest proximity to the given QoS. Then, select the Cl-State for the given P-State that satisfies the QoS. This selected configuration, P-State, Cl-State, is used for enforcing QoS strictness for each interval. This process is repeated across all cores.

In this study, we assume that the QoS is distributed homogeneously across all cores. In our case study, it is divided as 25% per core. At runtime, for the spawned threads, RePP samples thread behavior periodically and uses the models built to predict performance and power at all configurations. RePP then selects the optimal configuration for each interval to satisfy QoS.

[QoS]: In validating RePP-C on multicore architectures, we do two types of experiments: one for validating QoS for power and the other for performance. We define 2 input parameters: (a) frequency of change, and (b) load (or power). The load offered by the threads is constant between two load changes, which can occur in intervals of 1, 6, or 9 seconds. When the load changes, it can change based on a change_factor, as follows. Load, measured in MIPS, starts at a minimum, and varies by multiplying load by change_factor until it reaches a maximum load; thereafter, the load is multiplied by the negative value of change_factor until it reaches the minimum. The values of change_factor tested were 20% (**Turtle**), 35% (**Humane**), and 50% (**Devil**). The minimum load is defined as the minimum IPS (instructions per second) among all 4 threads running at minimum frequency; similarly, maximum load is the sum of IPS for all 4 threads at maximum frequency. In another set of experiments, we change the power consumed by the workload similarly to the load offered by the workload.

A violation in QoS occurs when RePP-C selects a configuration that makes the thread fall short of the minimum required IPS (or exceed the maximum power requirement) in a given mapping interval.

We ran ten experiments for power and ten for performance. Nine of them come from the combinations of the 2 parameters (change frequency and load/power) described above; the tenth comes from a **Random** setting within fixed boundaries of QoS. Selecting a broad spectrum of load (or power) and frequency of change allowed us to validate RePP-C across multiple combinations of configuration at runtime. Hereafter, we refer to Turtle, Humane, Devil and Random as **QoS types**.

A. RePP-C

In building performance and power models with core consolidation, the models need to address the following issues: (a) How to consolidate cores? (b) Which thread co-runners will minimize interference between workloads?; (c) What is the impact on power consumption and performance degradation?; (d) What is the computational latency to predict?

The technique of core consolidation can be enforced at runtime by removing the core from the set of active cores [23]. It can also be achieved by using C-States (This reduces the total power consumption of the inactive core to the baseline static power.) and ensuring that there exists at least one inactive core out of the set of active cores. In the remainder of the paper, the core consolidation mechanism is implemented using C-States, and we maintain the core in deep sleep state [24].

TABLE I: Component definitions for Intel Core i7

Component	Activity Formula	Modeled components
L1	perf_count_hw_cache_l1d	ld/st execution, mob, L1, L1 dtlb, L2 dtlb
L2	l2_rqsts:0xc0/ cpu_clk_unhalted:core_p	L2
LLC	last_level_cache_references/ cpu_clk_unhalted:core_p	LLC
MEM	perf_count_hw_bus_cycles/ cpu_clk_unhalted bus	memory, front side bus (fsb)

We verify the sleep state residency using the available machine specific registers (*MSR*) (Section V-A).

We build a training set consisting of workloads with varying memory footprint sizes and instruction rate from SPEC suite (Section V-B). The workloads are run for 100 seconds to gather PMCs and are sampled at every 250ms. The resulting model is validated using four other workloads which were *not* a part of training set (validation set) at different QoS types. These models are based on the activity recorded in the microarchitectural components such as L1, L2, LLC and MEM. A summary of the microarchitectural components can be found in Table I [25].

Intuitively, as the total number of active cores allocated to a workload decreases, the performance per thread and power consumption decreases. Contrary to our intuition, our findings show that only corunning threads have a negative impact on their performance and power consumption significantly, where as the reminder of the threads show minimal changes in performance and power consumption.

We implement FACTS to allocate threads-to-cores (at a reassignment interval of 1 second) while using RePP or RePP-C to select a configuration that ensure QoS guarantees. RePP-C is used to predict performance and power while having the total number of active cores less than the number of benchmark threads running on the system. Since the goal of RePP-C is to predict with core consolidation, we assume that there exists at least one inactive core. The estimated decrease in power consumption and performance due to the core consolidation is modeled offline using multilinear regression techniques (Eq. 1). The result, performance or power, is subtracted for every prediction on a per-thread level for the pair of co-running threads.

We build one model for performance and another for power. The models are built using a five step procedure:

(1) The performance and power limits are set exclusively under different QoS type for each workload.

(2) RePP is used to select a core configuration, P-State and Cl-State.

(3) Spawn the training workloads (Section V-B) consecutively using the scheduling policy FACTS.

(4) Gather PMCs and compute activity ratios for private L1, private L2, LLC and MEM for each thread.

(5) The difference between the PMCs read (or RAPL register (Section V-A) and per-thread performance (or power) limit for each thread that is corun in the workload, models the error due to core consolidation and co-runners.

Eq. 1 represents the multilinear regression model to predict the reduction in performance or power consumed due to core consolidation, where Δ_i represents the coefficient to be learnt and AC_i represents the activity ratio of the individual components (specifically L1, L2, LLC and MEM).

$$core\_consolidation = \sum_{i=0}^{comps} (\Delta_i \times AC_i) + constant \quad \text{(Eq. 1)}$$

The initial thread-to-core assignment is carried out statically, and the cores are set to the lowest frequency. Then, RePP predicts performance-power configurations that satisfy the performance and power requirement as given by the QoS. Thereafter, FACTS selects the potential co-runners for each thread running on the system, assuming that there exists at least one inactive core. In our case study, the core on which the thread with highest number of cache misses is allocated is the potential inactive core. Consequently, RePP-C is used to predict performance-power for each of the co-running threads, that is subtracting the value obtained in Eq. 1 from the original prediction (RePP). Thereafter, select RePP or RePP-C (Section IV-B).

B. Selecting RePP or RePP-C:

At runtime, both RePP and RePP-C, predict performance and power at all configurations for each thread. Selecting a configuration at every interval (1s) that satisfies the given performance or power QoS targets is based on (a) Favoring a scheduling algorithm for power or performance target exclusively; (b) Satisfying the system level QoS target; (c) Fulfilling the per thread QoS target.

The mechanism, that is used in the system is as follows:

(1) Select a configuration that satisfies per-thread target in both algorithms.

(2) Condition 1 (C1): Compute the error between total predicted output and QoS target.

(3) Condition 2 (C2): Compute the error between the per-thread predicted output and the QoS target per-thread.

(4) If C1 and C2 are approximately met for RePP and RePP-C, then select configuration as given using RePP, because it delivers a higher energy efficiency than RePP-C.

(5) If C1 is approximately met for RePP, but C2 for RePP-C or otherwise then: Select the configuration from either RePP or RePP-C based on how often one does better than other in satisfying per-thread QoS target.

V. Evaluation

In this section, firstly, we describe the experimental setup. Secondly, we describe the workloads used. Thirdly, we evaluate FACTS and compare against other state-of-the-art schedulers. Then, we validate RePP-C for a range of workloads, which are not a part of training, by satisfying the power and performance QoS targets. Finally, we discuss the scalability of the proposed technique.

TABLE II: Composition of **training** and validation workloads from the SPECcpu2006 benchmark suite.

Workload	Set of benchmarks
4C	Calculix Gobmk Bzip2 Tonto
2M2C	Lbm Astar CactusADM Calculix
4M	Astar Mcf Lbm Milc
3C1M	Povray Bzip2 Calculix Soplex
1C3M	Tonto Lbm Milc Soplex
4R1	Astar Bzip2 Mcf Soplex
4R2	Lbm Bzip2 Calculix GemsFDTD

TABLE III: Composition of workloads from the SPECcpu2006 benchmark suite for validating *FACTS*.

Workload	Set of benchmarks
8C	bzip2, calculix, gobmk, soplex, cactusADM, astar, tonto, povray
2M6C	astar, calculix, cactusADM, gobmk, soplex, povray, tonto, bzip2
4M4C	astar, milc, cactusADM, lbm, gobmk, tonto, povray, bzip
6M2C	xalancbmk, wrf, gemsFDTD, astar, lbm, tonto, calculix, cactusADM
8M	astar, tonto, cactusADM, gemsFDTD, lbm, milc, bzip2, mcf

A. Experimental Setup

We ran the experiments on an Intel SandyBridge processor with four cores enabled running Linux kernel version 3.14.5. Each core has a 64 KB on-chip private L1 cache and 256 KB private L2 cache. The total shared LLC was 6 MB, and the DRAM capacity of 8 GB. The processor is capable of frequency scaling from 0.8 GHz to 2.4 GHz. Turbo boost was disabled. The total number of Cl-States in this study are 50 per core. We do not enable any other internal thermal/power management algorithms which are run by the firmware.

We initialize one session of the performance monitoring tool perf per thread to characterize the thread behavior and periodically collect PMCs on a per thread session. We compile 25 SPEC benchmarks with gcc 4.8.1 and the Linux system call *sched_setaffinity* ensures that the thread runs on the specified core. The shortest thread in execution is always repeated to the end of the experiment. To minimize the noise as much as possible, we calculate the Percentage Absolute Average Error (**PAAE**) over a period of 100 seconds.

In addition to PMCs used in Table I, we also collect the PMCs instructions and cache-misses using perf for gathering the number of instructions retired and last level cache-misses to determine instructions retired and the cache-intensity of workloads.

We change P-States by setting the *CPUFreq governor* to *userspace* and scaling the frequency. Specifically, the MSR register MSR_IA32_PERF_CTL [25], [26] is used to reduce the latency to a single register move (Switching P-State using the MSR register overrides the Linux kernel driver *acpi-cpufreq*). To verify the C-State residency during core consolidation, the MSR registers [25] MSR_PKG_C3_RESIDENCY, MSR_PKG_C6_RESIDENCY and MSR_PKG_C7_RESIDENCY are read periodically every second. The computational cost (in cycles) was computed using the RDTSC instruction.

B. Workloads

In validating FACTS, we design eight single threaded workload ranging from CPU- to memory-bound [2]. We allow twice as many benchmark threads as cores. Similarly, for validating REPP-C, we build four single threaded workloads ranging from CPU- to memory-bound and map one benchmark for every core available. The workloads are generated using the technique described by Sanchez et al [27].

Similar to prior work [3], [5]–[8], [16], [19] we have also chosen a subset of benchmarks from the SPEC suite to represent the effectiveness of our technique.

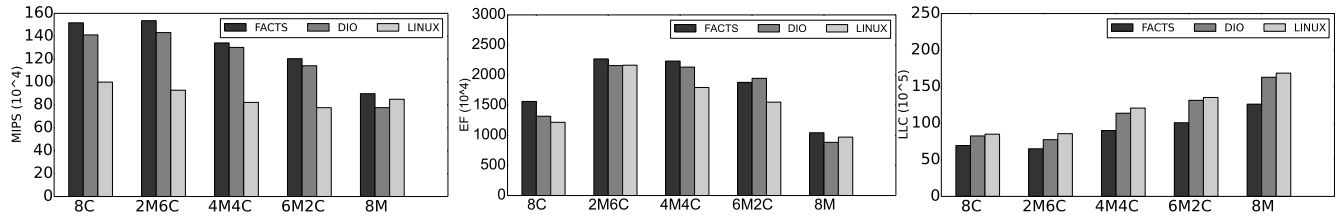
Table II refers to the workloads used in our environment for validating RePP-C. In particular, the boldfaced workloads are used for training the model: (**4C**) four compute intensive, (**2M2C**) two memory- and two compute-intensive, (**4M**) four memory intensive. The remaining workloads are used for validating: (3C1M) three compute- and one memory-intensive, (1C3M) one compute- and three memory-intensive, and (4R1, 4R2) two randomly chosen set of workloads consisting of four benchmarks each. Table III refers to the workloads used in our environment for validating FACTS. The legend for reading this Table is similar to Table II.

C. Validating FACTS and RePP-C

[FACTS]: In validating FACTS, DIO and Linux, we set for each scheduler all possible combination of P-States for the set of active cores in consecutive runs for every workload available. To compare the different scheduling policies, we analyze the results obtained for the total energy efficiency, activity in LLC and the total performance of all workloads for the *first 1000 seconds* of execution of the workload. Also, we study the pair of threads that are executed on a given core at a particular P-State. To ensure statistical significance of these results, each workload combination was run thrice and had a 95% confidence with very low error margin (less than 3%).

Previous research [3], [7], [19], [28] has shown that contention for shared resources in cache hierarchy can impact performance of the threads drastically. Miss penalty, measured in computational cycles, is defined as the ratio of memory access time to the inverse of frequency. Therefore, the power consumption increases as miss penalty to access the lower levels of caches increase.

Specifically, we show three main reasons why FACTS improves over DIO and Linux in performance and energy-efficiency when run for a fixed time quantum of 1000s. First, one might intuitively assume FACTS runs slower because it has a higher number of thread migrations which might cause high overhead and make the workloads take longer when dynamically mapped between cores, thus having less performance. Towards that, Fig. 1a shows that the total performance of FACTS improves over DIO by 8.25% (mean) and Linux by 37.56% (mean). Second, since FACTS runs threads faster than DIO, we assume that it might have a higher energy consumption. However, FACTS schedules workloads based on the demand for memory bandwidth and core P-States. Therefore, the threads which stall for memory are migrated



(a) Performance (MIPS) (b) Energy efficiency (MIPS/Watt) (c) LLC misses

Fig. 1: Performance, Energy efficiency and LLC misses in all workloads over all frequency set combinations for *FACTS*, *DIO* and Linux scheduler with 8 threads.

to cores with lower P-State, thus resulting in a small idle time for compute intensive threads and small power consumption while stalling for memory. Towards that, we looked at the energy efficiency (MIPS/watt) and show that FACTS improves over DIO by 6.2% (mean) and Linux by 14.17% (mean) (see Fig. 1b). Third, since FACTS migrates threads which stall for memory to cores with lower P-State, and DIO does not, it improves the performance of the co-scheduled threads and reduces the LLC misses by 4.87% (mean) over DIO and 41.70% (mean) over Linux (see Fig. 1c).

For instance, while executing workload 4M4C at P-States 1.6GHz, 2.4GHz, 0.8GHz and 0.8GHz on cores 0-3, respectively. DIO schedules cactusADM and Tonto on core with P-State 0.8GHz for 82% of the execution time. We note that cactusADM is a mid memory intensive, whereas, Tonto has low memory activity. On the other hand, FACTS schedules the pair of threads, Astar and Bzip2 are allocated for 10.65%; Astar and cactusADM for 15.2%; Bzip2 and cactusADM are allocated for 74.15% of the execution time on the core with P-State 0.8GHz. Specifically in this scenario, we improve over DIO by 6.84% in performance.

[RePP-C]: In validating RePP-C, we first demonstrate an example of the prediction technique at runtime, and then we evaluate RePP or RePP-C, in combination with FACTS to map threads to cores. This latter evaluation is done based on meeting the QoS target. See Section IV-B about how to select the use of RePP or RePP-C.

Fig. 2 shows an example of the power prediction technique implemented on our system for the first 20 seconds of execution for the workload 4R1 under QoS type Random. From top-to-bottom, the first graph represents the power as measured using *RAPL*, QoS target (Target) and the prediction made using *RePP-C*. The second and third graph show the changes in P-, Cl-States for all cores in response to changes in QoS target along the course of the execution. The fourth graph shows, if RePP (*non-consolidated*) or RePP-C (*consolidated*) is deployed. Finally, the fifth graph shows the corunners to the three active cores for RePP-C and four active cores for RePP. We highlight two results. First, RePP-C does show the capability to adapt to workloads consisting of multiple thread phases [2] and ensure QoS guarantees per-thread. Despite that fact that there are phases where RePP-C does not meet the QoS target, for instance, in seconds 7-10 RePP-C makes a

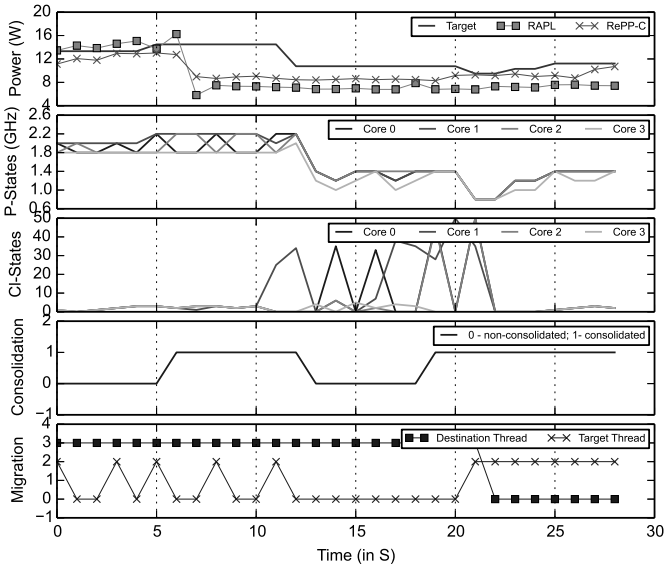


Fig. 2: Power prediction at runtime for workload 4R1 under QoS type Random.

3.6W error, the average error is as low as 4.03%. Second, RePP-C and RePP can predict power per thread which native RAPL or PEPP [16] cannot achieve. This shows that the model is accurate enough to capture the real behavior, is driven by existing performance counters, and, since the computational complexity at runtime is low, it can be used for fine-grain power management.

Fig. 3 shows the energy consumption in millijoules (mJ) using the power measured using native RAPL register and QoS target over time (Target) for four workloads under all QoS types for a period of 100 seconds. The X-axis represents the type of workload, QoS type and the frequency of change (in seconds). The configuration to satisfy the target is selected using our prediction technique, RePP-C. On average, under all QoS type the workloads exhibit an error in prediction of 7.55% or 115.67mJ. Observe, that the maximum error we incur is 286.4mJ (in 3C1M-Turtle_6). Moreover, the error in achieving the given energy target for QoS types Devil, Humane, Turtle and Randomized are 217.22mJ (16.23%), 41.63mJ (2.24%), 107.80mJ (5.69%) and 56.72mJ (3.02%) respectively.

Fig. 4 shows the total performance (MIPS) in thousands

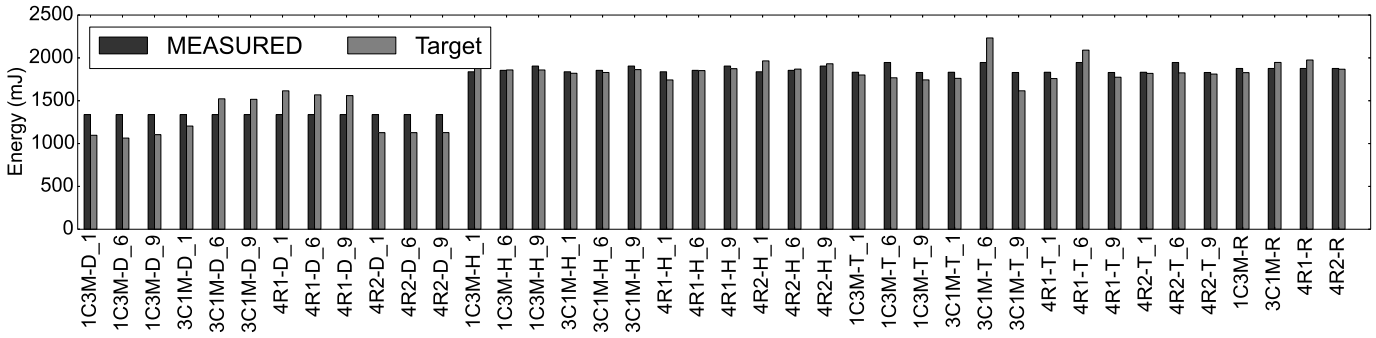


Fig. 3: **Energy consumed (mJ)** across all workloads under QoS type Devil (D), Humane (H), Turtle (T) and Randomized (R). The Y-axis is read as MEASURED *using RAPL* and QoS target over time (*Target*)

.

measured using PMCs and the QoS target over time (Target) for four workloads under all QoS types for a period of 100 seconds. The configuration ensuring QoS guarantees is selected using RePP-C. The X-axis represents the type of workload, QoS type and the frequency of change (in seconds). We incur an average error of 8.96% and a maximum error of 46.91% (in 4R2-Devil_9) and a total of 206.19MIPS.

Fig. 3 and 4 demonstrate that RePP-C has been validated under multiple QoS types with varying frequency of changes exhibiting different QoS requirements. Moreover, each histogram represents a summary of the workload execution, similar to Fig. 2.

[Scalability]: Scalability of RePP-C is based on the number of predictions per unit time possible for a given number of cores. In general, scalability of the prediction and scheduling techniques depends on computational capabilities (hardware configuration), the output required (number of configurations), the accuracy of the prediction technique, and the total time taken (in terms of computational cycles or ms) for producing the output. For RePP-C, we consider the following costs: (a) the cost to predict performance-power without consolidation (RePP), including reading PMCs and computing the activity ratio (b) the cost to predict performance-power with consolidation (RePP-C) (c) selecting RePP or RePP-C.

The computational cost (in cycles, refer Section V-A) to predict performance-power with RePP is 60,000 cycles and the consolidation incur an additional 5000 cycles. The cost to select RePP or RePP-C is 800 cycles at 2.4GHz.

The total number of configurations possible for a four core system are 656 billion (900 configurations per core for RePP and RePP-C) but due to the overlap in performance-power, prediction at all possible configurations leads to a redundancy in computational power. Therefore, we compute a total of 3600 configurations for a four core system. Observe that although that the total number of configurations predicted is negligible compared to total number possible, the models built predict performance-power with an accuracy greater than 91%.

Table IV shows the number of configurations that can be computed within a given quantum at a specific configuration. For example, the number of configurations for a 2 core system

TABLE IV: Number of configurations possible per core given the number of cores and time for modeling

Frequency	2.4GHz			0.8GHz		
Time(S)/#Core	1	2	4	1	2	4
0.500	900	230888	230888	900	76964	76964
0.250	900	115444	115444	900	38482	38482
0.125	900	57722	57722	900	19241	19241
0.063	900	29092	29092	900	9697	9697
0.031	900	14315	14315	900	4772	4772
0.016	900	7388	7388	900	2463	2463
0.008	900	3694	3694	900	1231	1231

are 810000 (900 configurations per core). However, given a P-State (say 2.4GHz) and a time constraint (say 0.125s), the total number of computable predictions are 57722 instead of 810000. Observe that for a single-core system at 2.4GHz or 0.8GHz, the number of configuration are same because there are only 450 configuration for each, RePP and RePP-C.

We show using Table IV that although the assessment of the prediction and scheduling policy has been carried out on a four core system, the computational time for predicting and scheduling workloads in manycore architecture takes a small time quantum and Table IV allows for determining the number of possible configurations that can predicted for a given time quantum at a specified frequency.

VI. Conclusion

The results from this paper are twofold. Firstly, we presented *FACTS*, an online frequency and contention-aware thread-to-core mapping scheduling policy for multicore architectures. FACTS can be implemented on any multicore processor without knowledge of the workload or the target system. Therefore, it is quick and easy to implement user-level Linux scheduler. We compared FACTS with other state-of-the-art schedulers and our results show improvements over all SPECcpu2006 workloads. FACTS improves energy efficiency by 6.2% and 14.17% over DIO and native Linux scheduler, respectively.

Secondly we extend RePP to RePP-C, a technique to predict performance and power at runtime parametrized by the hardware actuators, P-States and Cl-States in multicore

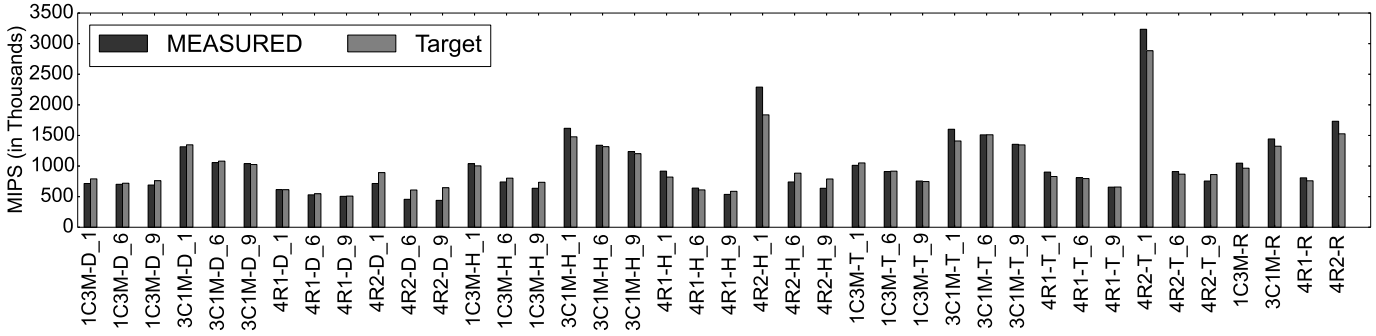


Fig. 4: **Total performance (in Thousands)** across all workloads under QoS type Devil (D), Humane (H), Turtle (T) and Randomized (R). The Y-axis is read as MEASURED *using PMCs* and QoS target over time (*Target*)

.

architectures to also predict in scenarios for workload consolidation. We implement FACTS in RePP-C for workload consolidation. The technique proposed includes a systematic methodology to build models to predict *performance-power* for each thread running on a multicore environment. The two prediction techniques (under performance limits and power limits) were built over *RePP-C* with multiple QoS targets and workloads have shown that the average errors of 7.55% and 8.96% (with 95% confidence interval) are observed when predicting energy and performance, respectively.

ACKNOWLEDGMENT

This work has been partially supported by the European Union FP7 program through the Mont-Blanc-2 project (FP7-ICT-610402), by the Ministerio de Economia y Competitividad under contract Computacion de Altas Prestaciones VII (TIN2015-65316-P), and the Departament d'Innovacio, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programacio i Entorns d'Execucio Paral.lels (2014-SGR-1051).

REFERENCES

[1] S. Blagodurov, S. Zhuravlev, et. al, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst. 2010*

[2] T. Kumar Prakash and L. Peng, "Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor," pp. 36 –41, 2008.

[3] A. Federova, S. Blagodurov, et. al, "Managing Contention for Shared Resources on Multicore Processors," *ACM commun., Feb 2010*

[4] R. Nishtala, M. Gonzalez, et. al, "A methodology to build models and predict performance-power in cmps," in *Int'l Workshop on Power-aware Algorithms, Systems, and Architectures (ICPP PASA) 2015*

[5] J. Mars, L. Tang, et al, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)-2011*

[6] J. Mars, L. Tang, et al, "Directly characterizing cross core interference through contention synthesis," in *Proc. of the Int'l Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC) 2011*

[7] R. Nishtala, D. Mossé, et al, "Energy-aware Thread Co-location in Heterogeneous Multicore Processors," in *Proc. of the Int'l Conf. on Embedded Software (EMSOFT) 2013*

[8] A. Gandhi, V. Gupta, et. al, "Optimality analysis of energy-performance trade-off for server farm management," *Perform. Eval. 2010*

[9] F. Bellosa, "The Benefits of Event: Driven Energy Accounting in Power-sensitive Systems," in *Proc. of Workshop on ACM SIGOPS European Workshop: Beyond the PC*, 2000.

[10] C. Isci and M. Martonosi, "Phase characterization for power: evaluating control-flow-based and event-counter-based techniques," in *Proc. of Int'l Symp. on High-Performance Comp. Arch. (HPCA) 2006*.

[11] ——, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)-2003*

[12] J. C. McCullough, Y. Agarwal, et al, "Evaluating the effectiveness of model-based power characterization," in *Proc. of the USENIX Conference on Annual Technical Conference (USENIXATC) 2011*

[13] K. Singh and M. Bhadauria, et al., "Real Time Power Estimation and Thread Scheduling via Performance Counters," *SIGARCH Comput. Archit. News*.

[14] A. Lewis and J. Simon, et al., "Chaotic Attractor Prediction for Server Run-time Energy Consumption," in *Proc of Power Aware Computing and Systems (HotPower) 2010*.

[15] R. Bertran, M. Gonzalez, et. al, "A Systematic Methodology to Generate Decomposable and Responsive Power Models for CMPs," *IEEE Trans. on Computers 2013*

[16] B. Su, J. Gu, L. Shen, et al, "Ppep: Online performance, power, and energy prediction framework and dvfs space exploration," in *Proc. of the Int'l Symposium on Microarchitecture (MICRO) 2014*

[17] S. Srinivasan, L. Zhao, et. al, "Efficient interaction between os and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev. 2011*

[18] R. Cochran, C. Hankendi, et al, "Pack & cap: Adaptive dvfs and thread packing under power caps," in *Proc. of the Annual Int'l Symp. on Microarchitecture (MICRO) 2011*

[19] S. Zhuravlev, J. C. Saez, et al, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv. 2011*

[20] B. Su, J. L. Greathouse, et al., "Implementing a Leading Loads Performance Predictor on Commodity Processors," in *Proc. of the USENIX Conference on Annual Technical Conference (USENIXATC) 2011*

[21] R. Miftakhutdinov and E. Ebrahim et al., "Predicting Performance Impact of DVFS for Realistic Memory Systems," in *Proc. of Int'l Symp. of Microarchitecture (MICRO) 2012*

[22] T. M. Le. (2009) A study on linux kernel scheduler version 2.6.32. [Online]. Available: http://goo.gl/UsYRms

[23] L. Kernel. sysfs devices. [Online]. Available: https://goo.gl/hBg3Du

[24] C. M. Olsen and C. Narayanaswami, "Powernap: An efficient power management scheme for mobile devices," *IEEE Trans. on Mobile Computing 2006*

[25] *Intel 64 and IA-32 Architecture Software Developer's Manual*.

[26] A. L. Brown, "The state of ACPI in the linux kernel," *SIGARCH Comput. Archit. News*, [Online]. Available: https://goo.gl/ZOMJHg

[27] D. Sanchez and C. Kozyrakis. "Vantage: Scalable and Efficient Fine-grain Cache Partitioning". *SIGARCH Comput. Archit. News*.

[28] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures," in *Proc. of the Int'l Conf. on Computing Frontiers (CF) 2006*