Energy Optimising Methodologies On Heterogeneous Data Centres

Rajiv Nishtala

Department of Computer Architecture
Universitat Politécnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

September 2017

Energy Optimising Methodologies On Heterogeneous Data Centres

by
Rajiv Nishtala

A Dissertation
Presented to the Department of Computer Architecture
at
Universitat Politécnica de Catalunya
in Candidacy for the Degree of
Doctor of Philosophy.

Thesis Advisory Committee

. . . . . .
Prof. Xavier Martorell Bofill, PhD.
Universitat Politécnica de Catalunya, Spain

. . . . . .
Prof. Daniel Mossé, PhD.
University of Pittsburgh, USA

Barcelona, September, 2017

Energy Optimising Methodologies On Heterogeneous Data Centres

Abstract

This is where you write your abstract ...

Contents

List of Figures

List of Tables

Todo list

Publication list

Contributions present in this thesis have been previously presented in the following proceedings:

1. **R. Nishtala**, Marc Gonzalez Tallada, Xavier Martorell, “A Methodology Methodology to Build Models and Predict Performance-Power in CMPs", in Proceedings of the 44th International Conference on Parallel Processing Workshops, ICPPW 2015, Beijing, China, September 1-4, 2015, pp. 193-202

2. **R. Nishtala**, Xavier Martorell, “RePP-C: Runtime Estimation of Performance-Power with Workload Consolidation in CMPs", in Proceedings of the 7th International Green and Sustainable Computing Conference, IGSC 2016, Hangzhou, China, November 7-9, 2016, pp.**NEED TO FILL THIS**

3. **R. Nishtala**, Xavier Martorell, Vinicius Petrucci, Daniel Mossé, “REPP-H: Runtime Estimation of Power and Performance on Heterogeneous Data Centers", in Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2016, Los Angeles, USA, October 26-28, 2016, pp.**NEED TO FILL THIS**

The following publications are under review:

1. **R. Nishtala**, Paul Carpenter, Xavier Martorell, Vincius Petrucci, “Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads", in Proceedings of 23rd IEEE Symposium on High Performance Computer Architecture, HPCA, 2017, Austin, USA, February 4-8, 2017, pp.**NEED TO FILL THIS**

2. **R. Nishtala**, Paul Carpenter, Xavier Martorell, “REPP: A Methodology for Runtime Estimation of Performance–Power in CMPs", in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2017, San Francisco, USA, April 23-25, 2017, pp.**NEED TO FILL THIS**

The following publications have been published, but not included in this thesis:

1. **R. Nishtala**, Daniel Mossé, Vinicius Petrucci, “Energy-aware thread co-location in heterogeneous multicore processors", in Proceedings of the Eleventh ACM International Conference on Embedded Software, EMSOFT, 2013, Montreal, Canada, September 29 - October 4, 2013, pp.21:1–21:9

Dedicated in loving memory of my grandfather, Rajeswara Rao Nishtala (PhD)

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Rajiv Nishtala
September 2017

Acknowledgements

I would like to dedicate this thesis to my struggle

Nomenclature

Latin Letters

$c_n = \arg\max_c R(w_n, c)$ The action with the highest reward, c_n among the possible set of actions, c, for the current state, w_n

c_n The action chosen from a finite set of alternatives for the current state, w_n, in time interval t_n to t_{n+1}

F Cache Friendly batch workload

N Insensitive batch workload

$Power_{\text{reward}}$ The ratio of TDP to

QoS_{curr} Currently measured tail latency at the 95th or 99th percentile for the latency-critical workload

QoS_{D} Defines danger zone as a co-efficient between 0 and 1

QoS_{reward} The ratio of QoS_{current} to QoS_{target}

QoS_{S} Defines safe zone as a co-efficient between 0 and QoS_{D}

QoS_{target} The target tail latency for the latency-critical workload

$R(w, c)$ A lookup table of rewards for state w, and action c

S Thrashing batch workload

T Cache Fitting batch workload

$\sum_{n=0}^{\infty} \gamma^n \phi_n$ The aim of MDP is to maximise this function, total discounted reward

w_n The current *"state"* in MDP refers to load of the latency-critical workload measured during time interval t_{n-1} to t_n

w_{n+1} The next *"state"* in MDP based on the current action, c_n

Greek Letters

α_c Current configuration, that is, (P_c, Cl_c)

α_f Future configuration, that is, (P_f, Cl_f)

χ Load or power in the step-wise monotonic function

$\chi(\kappa)$ κ^{th} element in the data sequence for either load or power

$\chi(min), \chi(max)$ The minimum, maximum value of load or power

Cl_c Current Cl-State

Cl_f Future Cl-State

Cl_i Intermediate Cl-State

Δ, β Co-efficients in the multi linear regression model

η_c Performance (MIPS) at current configuration

η_f Performance (MIPS) at future configuration

γ Discounting factor in MDP

P_c Current frequency for P-State

P_f Future frequency for P-State

ϕ_n Positive or negative reward in MDP for the chosen action, c_n is updated at t_{n+1}

P_i Intermediate frequency for P-State

ψ Number of datapoints before the peak in a step-wise monotonic function

ρ_c Power at current configuration

ρ_f Power at future configuration

τ Change_factor for the step-wise monotonic function

ξ Learning factor in MDP

Abbreviations

AAE Absolute Average Error

AR Activity Ratio

Big ARM Cortex-A57

BPU Branch Predictor Unit

CFS Completely Fair Scheduler

$Cl-States$ Idle states

DPM Dynamic Power Management

$DVFS/P-States$ Dynamic Voltage, and Frequency Scaling

FE Front End

FP Floating Point Unit

HipsterCo A variant of Hipster that optimises improving batch workload throughput while meeting deadlines for latency-critical workloads

HipsterIn A variant of Hipster that optimises energy reduction while meeting deadline when latency-critical workloads are running solo

INT Integer Unit

IPS Instructions Per Second

*L*1 L1 Cache Memory

*L*2 L2 Cache Memory

LLC Last Level Cache

MDP Markov Decision Process

MSR Machine Specific Registers

PAAE Percentage Absolute Average Error

PUE Power Usage Effectiveness

QoS Quality of Service

QPS Queries per Second

RAPL Running Average Power Limit

REPP Runtime Estimation of Performance and Power

REPP − C Runtime Estimation of Performance and Power with workload consolidation

REPP − H Runtime Estimation of Performance and Power across Heterogeneous architecture

RL Reinforcement learning

RPS Requests per Second

Small ARM Cortex-A53

STDEV Standard Deviation

TDP Thermal Design Power

PART I:

PROLOGUE

Errors using inadequate data are much less than those using no data at all.

CHARLES BABBAGE

CHAPTER 1

Introduction

difference between supercomputing centres, and data centres test[1]

Importance of low end processors, and the extension of vector support for ARM processors (!) [1]

nishtala: need to speak about how latency-critical, and batch workloads are used

nishtala: need for power caping?

[1] In what follows, unless stated otherwise, I will use the words: I, My, We, Our refer to my exclusive contributions

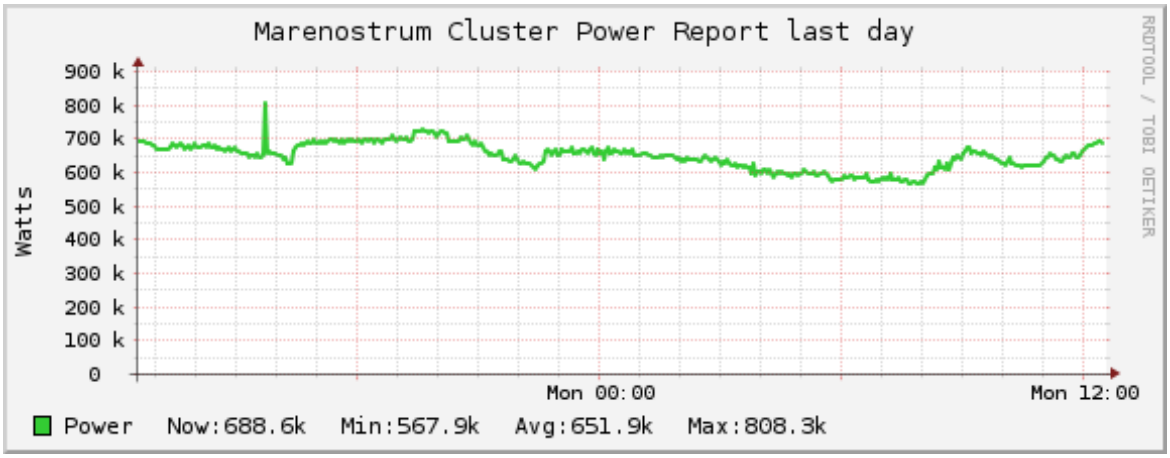


Figure 1.1 Power mare nostrum over the course of a day

CHAPTER 2

Background

CHAPTER 3

Infrastructure

nishtala: need a graph — moores law

THIS chapter gives a gist of the architectures, the power meters used, the type of applications, and their characterisation. In recent years, the multicore architectures deployed in server systems are increasingly visible in both mobile processors (for example: the ARM Juno, and AppliedMicro XGene), and supercomputers such as `MareNostrum` that they have become a norm for modern 64 bit multicore systems. Such non-restrictive deployment of multicore architectures in large-scale data centres led to a great divide between power and performance, which are tightly bound. This great divide makes it increasingly difficult to find a balance between the two. For instance, the `MontBlanc` project [2–4] from the European union deployed multiple mobile cores to build a supercomputer which is "power efficient" (performance per watt). On the one hand, delivering higher performance implies consuming more power; On the other hand, delivering lower performance might imply missing a deadline for a latency-critical workload (such as a bank transaction or web-request which tend to terminate within milli/micro seconds [5, 6]), while saving some power. In the recent years, several new processors [7–12] have been proposed in search of the optimal point in the "pareto curve", thereby making an algorithm generalised if it works across multiple architectures and processors.

3.1 Architecture

nishtala: will Hipster and repp be explained by now?

In this section, we present the architectures and processors our contributions have been evaluated on, unless otherwise stated.

Intel Intel Corei7 Sandy Bridge processor [13] has four out-of-order cores enabled, each with 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 with Linux (kernel 3.14.5). The processor is capable of Dynamic Voltage and Frequency Scaling (DVFS/P-States) 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.

AMD AMD Phenom II [14] processor has four out-of-order cores enabled, each with 64 KB on-chip private L1 cache and 512 KB private L2 cache. The total shared LLC was 6 MB, and the DRAM capacity of 8 GB with Linux (kernel 3.13). The processor is capable of DVFS from 0.8 GHz to 3.2 GHz. AMD processor has no Cl-States. We do not enable any other internal thermal/power management algorithms which are run by the firmware.

ARM ARM Juno R1 developer board [15] has Linux (kernel 4.3). The Juno board is a 64 bit ARMv8 big.LITTLE architecture with two high-performance out-of-order Cortex-A57 (big) cores and four low-power in-order Cortex-A53 (small) cores. The cores are integrated on a single chip with off-chip 8 GB DRAM. The two big cores form a cluster with a shared 2 MB L2 cache, and the four small cores form another cluster with a shared 1 MB L2 cache. The big cores are capable of DVFS from 0.6 GHz up to 1.15 GHz, whereas the small cores are fixed at 0.65 GHz. A cache coherent interconnect (CoreLink CCI-400) provides full cache coherency among the heterogeneous cores, allowing a shared memory application to run on both clusters simultaneously.

X-Gene2 AppliedMicro X-Gene2 [16] board has Linux (kernel 4.1). The X-Gene2 is a 64 bit homogeneous architecture with eight out-of-order ARMv8-A cores enabled at 2.4 GHz with 128 GB DRAM.

3.2 Power Meters

To gather power measurements, we use the machine's power sensor (in case of Intel and ARM) or an external power meter (in case of AMD as power sensors are not available) periodically during execution. The X-Gene2 board is equipped with neither a power sensor nor attached to a power meter.

The power meters used are as follows. (1) The *RAPL* (running average power limit) register in the Intel architecture records power of core, uncore, and DRAM controller [17]. The RAPL interface is accessed using the Machine Specific Registers (MSR) available on Intel processors [18]. (2) *WattsUP pro* for AMD: external device that records power of the entire system (power from outlet) [19]. (3) *Four native energy meters* for ARM [20, 21]. These registers report separately the power consumed by the big cluster, small cluster, and the rest of the system (Juno's *sys* register [22]). The power consumption of the Mali GPU is also available, but it is negligible because the GPU is disabled in all our experiments.

Table 3.3 summarises the architecture, processor, Linux kernel version, the power meters, the compiler (gcc) version used for compiling the benchmarks, the range of core frequencies (min-max) when using DVFS, the L2 and L3 cache sizes (ARM does not have L3). The machines were available across multiple institutions (Barcelona Supercomputing Center, University of Pittsburgh, and Universidade Federal da Bahia), and therefore we had no control over the kernel and gcc versions; regardless, the results for each architecture are consistent within that set of experiments. In what follows, I refer to Intel Sandybridge as Intel, AMD Phenom II as AMD, and ARM Juno R1 as ARM.

nishtala: need to figure out where this section goes... becuase we speak about latency critical workloads

3.3 Performance Monitoring

We use performance monitoring tool, `perf` [23], on all architectures to gather `perf_events`. Alternatives to perf include the profiling tools [24] supported by Docker, Kubernetes and LXC [25].

The ARM Juno board raises two challenges while gathering PMCs: (1) It does not allow counters other than instructions, cache-misses, and cycles to be read for Cortex A53, the in-order processor, as the `CPTR_EL3` (Architectural Feature Trap Register) is not implemented in the Juno chip, thereby limiting the scenarios it can be used. (2) There is a known bug [26] that causes perf to generate garbage values for all cores whenever any core enters an idle state. Since performance statistics are only needed when batch workloads are run, we overcome this by disabling CPUidle [27, 28]. This prevents Linux from entering the cores in an idle state when changes in the mapping cause idle periods longer than 3500 µs.

Since `CPTR_EL3` is not implemented on Cortex-A53, it is used only to gather basic performance statistics when batch workloads are collocated with latency-critical workloads. In other words, without collocation multiprogrammed batch workloads are run only on Cortex-A57 on the ARM processor.

The characterisation of workload is achieved by periodically collecting performance statistics from the latency-critical and batch workloads. For the latency-critical workload, we gather the appropriate application-level QoS metrics such as throughput (Requests Per Second – RPS or Queries Per Second – QPS) and latency (query tail latency). For the batch workload, we use `perf` to characterise the thread behaviour using the perf per-thread session.

3.4 Workloads

In recent years, data centres, and mobile architectures [2–4, 29–34] have increasingly observed large number of workloads of different types, which stress the activity in different microarchitectural components. Specifically, we have chosen two classes workloads: batch workloads or throughput-oriented workloads [35, 36], and interactive or latency-critical workloads [29–34]. These workload classes are interesting due to their inherent differences in the interaction with the microarchitectural components, and service level agreements that the applications need to maintain.

nishtala: Should the workloads be written in the appendix?

3.4.1 Batch workloads

We categorise the batch workloads into single threaded batch workloads, and multiprogrammed batch workloads. For single-threaded batch workloads, we ran 22 SPEC CPU 2006 [37, 38], nine PARSEC 3.0 [39], six NAS [40], 11 SPLASH2x [41]. These workloads exhibit both memory, and computational bound phases. The number of benchmarks in a multiprogrammed workload is equal to the number of cores in each architecture, thereby, having 35 workloads of four benchmarks on AMD and Intel, and ten workloads of two benchmarks on ARM, based on the methodology described by Sanchez and Kozyrakis [42].

The benchmarks are divided into four categories, following the categorisation by Sanchez and Kozyrakis [42]. We run all the applications in isolation and gather performance monitoring counters (PMCs) at a sampling frequency of 250 ms until the end, and compute mean value of the ratio of number of kilo-LLC misses (LLC: Last Level Cache) for kilo-instructions-per-second (kilo-IPS). For all workloads, we select the native input set size. Those which have ratio less 0.1 are considered as Thrashing (**S**), applications that benefit cache size, that is, ratio between 0.1 and 0.2 are classified as Cache-Fitting (**T**). Then, applications with a ratio between than 0.2 and 1 are classified as Cache-Friendly (**F**). Finally, applications with ratio greater than 1 are classified as Insensitive (**N**). There are 35 possible combinations (with repetitions) of these four categories, each of which forms a group. In the multiprogrammed

workloads consisting of four benchmarks (on Intel, and AMD) from SPEC, PARSEC 3.0, NAS, and SPLASH2x suites, we have one mix per group. In the multiprogrammed workloads, each application is randomly selected from the category. This results in 35 workloads. The same methodology is followed on ARM, where there are 10 possible combinations (with repetitions) of these four categories, each of which forms a group. In those multiprogrammed workloads consisting of two benchmarks from PARSEC 3.0, NAS, and SPLASH2x, we have one mix per group. Note that SPEC could not be compiled on ARM. Table **??** shows the categorisation of the workloads. This technique of selecting multiprogrammed workloads facilitates for significant difference in the size of the shared memory footprint and number of stall cycles of the processor.

3.4.2 Interactive workloads

nishtala: need not be called contributions.. if REPP and HIPSTER are defined before

We evaluate the effectiveness of the contributions using two interactive workloads, Memcached [43], and Web-Search [44], which have distinct characteristics and impact on shared resources [30]. **Memcached** [43] is an open source implementation of an in-memory key-value store for data caching used in many services from Twitter, Facebook and Google [5, 6]. The backend of **Web-Search** [45, 46] is an instance of Elasticsearch [44], an open source implementation of a search engine used by many companies including Netflix and Facebook. The load generator (Faban [47]) for Memcached and Web-Search is adapted from CloudSuite 3.0 [48]. It is configured to model diurnal load changes (Figure **FIG NUMBER**)

nishtala: diurnal load figure number

, simulating a period of 36 hours [34]; each hour in the original workload corresponds to one minute in our experiments.

We also evaluate the effectiveness by collocating a single latency-critical workload, and a mix of batch workloads. The objective of the collocation is to maximise the throughput of the batch workloads while satisfying QoS of the interactive workload. The number of batch workloads is equal to the number of cores not utilized by the latency-critical workload. We report the system throughput by aggregating the IPS of all batch programs.

nishtala: Hipster is mentioned here so maybe think about how you will satisfy this scenario

When implementing Hipster, the ARM processor is used either when collocating interactive workload with batch workload, or running an interactive workload in isolation, while the AppliedMicro XGene2 machine is allocated exclusively for the workload generator.

Table 3.1 Latency-critical workload configuration. Workload configurations, maximum load while meeting the target tail latency with two big cores.

App	Workload Configuration	Max. Load	Target Tail latency (ms)
Memcached	Twitter caching server of 1.3 GB	36 000 RPS	10 (95%ile)
Web-Search	English Wikipedia Zipfian distribution	44 QPS ǀ think-time of 2sec[47]	500 (90%ile)

Table 3.2 Power and performance characterization on Juno platform. **–need to fix this**

	$Power\ (Watts)$		$Perf.(IPS \times 10^6)$	
Core type (GHz)	All cores	One core	All cores	One core
Big A57 (1.15)	2.30	1.62	4,260	2,138
Small A53 (0.65)	1.43	0.95	3,298	826

Tail latency requirements — For Memcached, we define the tail latency to be the 95th percentile request latency, with a target of 10 ms [29, 48]; for Web-Search, we define it to be the 90th percentile query latency, with a target of 500 ms [48]. Table 3.1 lists the two latency-critical applications, their configurations, maximum loads, and target tail latency in milliseconds. For each latency-critical workload, the maximum load is chosen so that the platform is able to meet the tail latency when running on the big cores at maximum DVFS.

3.5 Conclusion

In this chapter we have introduced the architectures, the power meters, the type of workloads, and the way performance monitoring is carried out. In a nutshell, the work has been carried out on three 64 bit architectures: Intel SandyBridge, AMD Phenom II, and ARM Juno. Each of these architectures is equipped with a power meter to gather power statistics. Similarly, we installed a performance monitoring tool to gather performance statistics per application. Finally, the single-threaded workloads used were taken from four different benchmark suites: SPEC CPU 2006, PARSEC, SPLASH, and NAS. The multiprogrammed workloads were designed based on the methodology described by Sanchez and Kozyrakis [42].

3.6 Power efficiency

nishtala: need to add this to the text – also amd, and intel

The power consumption of the Juno platform is obtained by reading specific hardware registers [21]. These registers report separately the power consumed by the big cluster, small cluster, and the rest of the system (Juno's *sys* register [22]). The power consumption of the Mali GPU is also available, but it is negligible because the GPU is disabled in all our experiments. Performance is quantified using the IPS measured by hardware performance counters.

Table 3.2 summarizes the power and performance characteristics of the big cluster (2 cores) and the small cluster (4 cores). We characterize the heterogeneous platform by running a compute-intensive microbenchmark (same as used in Section 5.1.3) consisting of mathematical operations without memory accesses. We report the system power consumption as the sum of the big and small clusters and the rest of the system (including memory controllers, etc). For the per-cluster comparison, we run the microbenchmark on all the cores in the cluster (two big cores vs four small cores). For a per-core comparison, we run the microbenchmark either on a single big core or on a single small core.

Considering system power, a single big core is 52% more power-efficient than a single small core, in terms of IPS per watt (Table 3.2). However, taking into account all cores in a cluster, and assuming that all cores can be fully utilized, a small cluster is 25% more power-efficient than a big cluster. This discrepancy is due to the rest of the system, outside the core clusters, which consumes about the same power as a big core at full utilization (0.76 W). If we subtract the power of the rest of the system, a single small core is 2.3× more power-efficient than a big core. We notice that small cores are attractive for throughput-oriented workloads, because of improved power efficiency. Big cores are, however, still needed for latency-critical workloads with tight QoS constraints, as a result of computationally-intensive single-threaded requests.

Table 3.3 Machines used in this study.

Processor	Linux Kernel	Power Meter	gcc	DFS (in GHz)	L2 (size in kB)	L3 (size in MB)
Intel Core i7-2760QM (4 cores)	3.14.5	RAPL	4.8.1	0.8-2.4	256	6
AMD Phenom II X4 B97 (4 cores)	3.13.0	WattsUp Pro	4.9.2	0.8-3.2	512	6
ARM Juno 64 bit	4.3.0	Native energy	4.9.2	0.6-1.15 (Cortex A57)	2048 (Cortex A57)	no L3
(ARMv8 – 2 Cortex A57 and 4 Cortex A53)		meter		0.65 (Cortex A53)	1024 (Cortex A53)	
Applied X-Gene2 64 bit (8 cores)	4.10	no meter	5.2.0	2.4	256	8

		Platform		
Benchmark	*Suite*	*Intel*	*AMD*	*ARM*
Bzip2	SPEC	N	N	-
Calculix	SPEC	N	N	-
Gobmk	SPEC	N	N	-
Gromacs	SPEC	N	N	-
H264ref	SPEC	N	N	-
Hmmer	SPEC	N	N	-
Namd	SPEC	N	N	-
Omnetpp	SPEC	N	N	-
Povray	SPEC	N	N	-
Tonto	SPEC	N	N	-
CactusADM	SPEC	F	F	-
Lbm	SPEC	F	S	-
Libquantum	SPEC	F	S	-
Sjeng	SPEC	F	N	-
Zeusmp	SPEC	F	F	-
Wrf	SPEC	F	F	-
Astar	SPEC	T	T	-
Bwaves	SPEC	T	T	-
Soplex	SPEC	T	T	-
GemsFDTD	SPEC	T	S	-
Mcf	SPEC	S	S	-
Milc	SPEC	S	S	-
Xalancbmk	SPEC	S	S	-
ep.C	NAS	N	N	N
is.C	NAS	N	N	N
cg.C	NAS	F	F	S
lu.C	NAS	S	S	S
mg.C	NAS	S	S	S
sp.C	NAS	S	S	S
Blackscholes	PARSEC	N	N	N
Bodytrack	PARSEC	N	F	S
Dedup	PARSEC	N	T	S
Canneal	PARSEC	T	F	T
Facesim	PARSEC	N	N	F
Fluidanimate	PARSEC	N	N	N
Freqmine	PARSEC	N	N	N
Raytrace	PARSEC	N	N	T
Streamcluster	PARSEC	S	S	F
Vips	PARSEC	N	F	T
x264	PARSEC	F	F	T
Barnes	SPLASH2x	F	T	T
Fft	SPLASH2x	N	N	N
Fmm	SPLASH2x	F	F	S
lu_cb	SPLASH2x	N	N	N
lu_ncb	SPLASH2x	F	F	N
ocean_cp	SPLASH2x	F	F	T
radiosity	SPLASH2x	N	N	F
Radix	SPLASH2x	T	T	T
Raytrace	SPLASH2x	N	N	T
Volrend	SPLASH2x	N	N	F
Water_spatial	SPLASH2x	F	N	T
Water_nsquared	SPLASH2x	N	N	F

Table 3.4 Categorisation of workloads. The label, and description for each benchmark from SPEC, NAS, PARSEC, and SPLASH2x. The labels are read as follows: N as Insensitive, F as Cache-Friendly, T as Cache-Fitting, and S as Streaming/Thrashing

PART II:

PREDICTION AND MODELLING TECHNIQUE

> Errors using inadequate data are much less than those using no data at all.
>
> CHARLES BABBAGE

CHAPTER 4

REPP: Runtime Estimation of Performance–Power

nishtala: benchmark names are always *emphasized*

So far we have understood the infrastructure used in this dissertation: the architectures which are composed for different microarchitectural capabilities, and power-efficiencies, and those applications that make intensive use of different microarchitectural components on the modern data centre infrastructures, and aimed at workload throughput, or interactiveness to meet a certain deadline (or target). There also exists multiple other types of applications that require immediate access to resources. In this chapter, we describe our work on power and performance management while considering contention for CPU, as well as shared resources of the machine for throughput-oriented workloads.

Our prior work[1] [49], which focuses on energy-efficient (throughput per watt of energy consumed) cluster scheduling to map threads to cores using statistical information gathered from PMCs. In such scenarios applications are scheduled by either sharing a core with another application from a different class to improve energy efficiency (i.e., time sharing a core), or by running in isolation (i.e., no time sharing). Time sharing a core allows to consolidate workloads on a same server, thereby allowing data centre servers to save energy as they go underutilised for long durations of the day [50, 34, 29, 33, 30].

nishtala: PLOT GOOGLE FIGURE, and EXPLAIN

However, with the increasing demand for low cost, and low power high performance computing, multiple vendors such as Amazon EC2, and IBM Soft Layer, etc., provide users the capability to "rent" computing nodes to process their computational needs without the need for purchasing the equipment. In addition to providing low cost services, it is beneficial for vendors of modern data centres to run at maximum capacity because data centres consume as much idle power as they are running at 60% capacity, thereby driving

[1] Published as a part of my master thesis, in proceedings of EMSOFT 2013, and not included in this thesis.

computational services to run at maximum load. Therefore, to provide such low cost, and low power computing services, modern data centre architectures [51], incidentally or intentionally, have to deal with server architecture heterogeneity [52, 53]. Nevertheless, some critical challenges for such a heterogeneous data centre administration are (1) mapping applications across multiple architectures; (2) providing a service that is reliable; (3) availability on demand for a given computing service; (4) availability at an attractive cost for end-users.

Given these challenges in heterogeneous data centres, a service that can be delivered using native hardware capabilities may be represented in various forms: for instance minimising energy consumption subject to a performance target and peak power constraint, or maximising performance subject to a peak power constraint or to enable energy-efficient cluster management in data centres (which thread is assigned to each core and what frequency to set each core when). For instance, current data centre tools such as the Intel Intelligent Power Node Manager [54], the OpenStack framework [55] or the IBM Tivoli framework [56], enable power-thermal control, and optimisation like monitoring, and capping capabilities. Irrespective of the how the service is formulated, any solution, whether optimal or heuristic-based, requires a fast and accurate model to predict how a potential change in DVFS, C-State or other low-level power state would translate into real thread performance and power demand, and satisfy these constraints.

Traditionally such services have been met using iterative algorithms which are P-State based: they monitor the application behaviour history periodically [57] and set the DVFS configuration for the next quantum. But these approaches require multiple iterations before power and performance criteria are satisfied and can lead to massive violations in power and performance budgets for data centres [52, 58].

In response to traditional algorithms, fast and reactive prediction based algorithms provide the benefit of quick response and reaction for a small fraction of computation costs. The broad spectrum of computation, and memory behaviour can be understood by monitoring the PMCs, available on most commercial processors. For instance, applications that are memory bounded, do not take benefit of higher P-State as they stall for memory, thus using a lower P-State might be sufficient to satisfy the performance constraint while saving power. At the same time, the converse is true for compute bounded workloads. Using PMCs, we demonstrate that our prediction algorithm can determine if the application is scalable with DVFS at runtime and can select the most appropriate configuration given a constraint. Such techniques are also important to enable energy-efficient cluster management in data centres (which thread is assigned to each core and what frequency to set each core when).

Such services are provided using the dynamic power management features [59] (DPM) available on modern operating systems, which led to multiple low-level hardware capabilities

on modern processors. thereby making it very difficult to understand the combinatorial complexity they bring in terms of service comprehension even for reactive prediction based algorithms. Moreover, administrators of large-scale data centres have to deal with metrics such as performance, power and efficiency levels (e.g., IPS, IPS/W, or Power Usage Effectiveness (PUE)), which do not directly correspond to DVFS, Cl-States or core shutdown. For instance, on the Intel machine with four cores there are nine different DVFS states that can be combined with 50 clamping configurations (Cl-States) per core. This gives the administrator a total of 40 billion configurations, each one resulting in a particular performance and power level.

This complexity in hardware explains why current solutions operate at a coarse granularity, at node level, while the hardware features work at a core level, with low overheads. Understanding the behaviour at per-core or thread level allows for a precise control over performance power contribution and their corresponding effects.

nishtala: check if this is what you want to write?... still flow is missing

There is a need for application agnostic prediction approach in heterogeneous data centres to meet a criteria, irrespective of how it is formulated. This is precisely what our methodology provides.

In this Chapter, firstly we introduce *Runtime Estimation of Performance and Power*, **REPP** [60], a performance and power model for single-core Intel processors parametrised by P-States and Cl-States. REPP is a methodology to generate fast, and accurate performance and power predictions. The model is accurate enough to capture the real behaviour, is driven by existing performance counters, and, since the computational complexity at runtime is low, it can be used for fine-grain power management. As a result, irrespective of the formulation of the problem, the models built can used to optimise a given parameter.

Secondly, we introduce an extension for REPP to scale power and performance prediction techniques from single-threaded workload to a variety of multi-programmed workload across server architectures. We specifically designed a *Runtime Estimation of Performance and Power across Heterogeneous architectures*, **REPP-H** [61], to estimate and *control* power-performance on server architectures running multiple workloads using statistics gathered on real processors.

Finally, we extend REPP-H to scale power and performance prediction technique with workload consolidation. Specifically, we designed a *Runtime Estimation of Performance and Power with workload consolidation*, **REPP-C** [62], to estimate and control power and performance on server architecture (with consolidation) running multiple workloads.

The evaluation of REPP-H, and REPP-C, on each architecture, was carried out for single-threaded and multiprogrammed applications with multiple constraints, and we show that power capping mechanism and ensuring minimum performance is delivered are met.

Recapping the discussion, we introduce REPP, a methodology to build single core model for predicting power and performance (Section **ONE**) across P-State, and Cl-States. Next, these models are extended to work in a multi-core environment (Section **TWO**), and meeting power and performance constraints across multiple single-threaded, and multi-programmed workloads. Finally, these models are extended to work with workload consolidation which is increasingly common across modern data centres.

4.1 REPP: Runtime Estimation for Performance and Power

We propose Runtime Estimation for Performance and Power (*REPP*) in order to model and estimate performance and power on three major architectures: Intel, ARM or AMD. *REPP* is modelled using multi linear regression models, constructed in two steps: *offline* modelling and *online* modelling.

Offline modelling: In the offline modelling technique, a subset of applications from representative benchmarks (Section 4.1.1.1) suites are profiled to build to power (Section 4.1.1.2), and performance (Section 4.1.1.3) models offline for a single-core. The models built can predict power, and performance at the available P-State, and Cl-State with high accuracy. The offline design phase is attractive because (a) it eliminates the overhead of learning and tuning the performance and power model at numerous P-States and Cl-States; (b) it does not rely on using power sensors/meters to estimate power and performance at runtime; and (c) it is a one-time effort.

Online modelling: In the online modelling technique, we extend our models to the multi-core environment, by *aggregating* the per core contributions. The models exposed to the multi-core case, are modelled to tackle the contention due shared resources. Finally, we describe a technique for power, and performance control across multi-core environment.

In the remainder of the section, we focus on the the offline modelling procedure, and the offline, and online validation of the single-core models. The notations used hence-forth for the rest of the *chapter* are detailed in Table 4.1.

4.1.1 Single core offline modelling

We predict performance and power in a different P-State and Cl-State, based on the activity recorded in the microarchitectural components floating point units (FP), integer units (INT), front end (FE), branch predictor unit (BPU), private L1 cache (L1), private L2 cache (L2), last level cache (LLC) and the memory sub-system (MEM). Since these components do not have PMCs that directly record their activity, we use the available PMCs to calculate each component's **activity ratio**. The activity ratio is defined as the component's average number of µops per cycle (µops/cycle).

Table 4.1 Summary and description of the symbolic notation

Notation	Description
P_c	Current P-State.
P_i	Intermediate P-State.
P_f	Future P-State.
P_c	Current frequency for P-State.
P_i	Intermediate frequency for P-State.
P_f	Future frequency for P-State.
Cl_c	Current Cl-State.
Cl_i	Intermediate Cl-State.
Cl_f	Future Cl-State.
α_c	Current configuration, that is, (P_c, Cl_c).
α_f	Future configuration, that is, (P_f, Cl_f).
η_c	Performance (MIPS) at current configuration.
η_f	Performance (MIPS) at future configuration.
ρ_c	Power at current configuration.
ρ_f	Power at future configuration.

Table 4.2 summarises for Intel [13] the microarchitectural components, and modeled components. Similar components were also modeled on AMD, and ARM. Table 4.3 summarises the microarchitectural components, activity formulas for AMD, ARM and Intel, and the raw perf event registers [14, 15, 13]. To compute the activity ratio, the activity in each component, for each architecture, is divided by `cpu_clk_unhalted (r076)`, `cpu_cycles (cycles)` and `cpu_clk_unhalted:0x01 (r13c)`, respectively.

In building the multilinear regression models, we specifically profile the activity in the aforementioned microarchitectural components because our results using microbenchmarks on each architecture have shown that these microarchitectural components have a high dynamic power. We define dynamic power as the difference between the current power consumption and power consumption when idle. The activity formulas were built using carefully selected PMCs that are highly correlated with the dynamic power and performance. For instance, on the Intel platforms' pipeline functionality, the scheduler which is a part of the out-of-order engine has six issue ports, out of which ports 0, 1 and 5 are shared for integer, branch instructions and floating point instructions. However, there are counters for each port, branch instructions and floating point instructions. Therefore, we subtract the instructions issued using ports 0, 1 and 5 and the branch instructions and number of floating point instructions to get the total number of integer instructions. By contrast, on AMD and ARM, the microarchitectural components have unique counters for total number of integer instructions.

Table 4.2 Component definitions for Intel Core i7

Component	Modelled components
FE (IPC)	L1_ITLB, L1_ICACHE, PREDECODE, FETCH_UNIT, μCODE ROM, μOP BUFFER, LSD, SPT, RAT, ROB, RETIRE
INT	Integer arithmetic units
FP	Floating point arithmetic units
BPU	BPU and branch execution
L1	LD/ST execution, MOB, L1, L1 DTLB, L2 DTLB
L2	L2
LLC	LLC
MEM	Memory and Front Side Bus (FSB)

nishtala: make the component activity ratios capital!

done

4.1.1.1 Training the models

Our performance and power models for a single core are built using a random training set from representative benchmark suites (training data set). These benchmarks are executed to gather the PMCs values during a period of 100 seconds with a sampling period of 250 ms, at all P-States and a subset of the Cl-States. The models built are exclusive to a given architecture, and are rebuilt for each architecture REPP is implemented on.

For this study, the training data set consists of three floating point, and three integer benchmarks, which are chosen at random, and run at all P-States, and five Cl-States. This results in 270 experiments on Intel, 24 experiments on AMD, and 18 experiments on ARM. Recollect that AMD, and ARM do not have Cl-States. The Cl-States are chosen such that there is at least 10% variation (specifically the Cl-States 10, 20, 30, 40, 50). To validate the resulting single-core models at runtime/online, we use the remainder of the benchmarks from SPEC CPU 2006, PARSEC3.0, NAS, and SPLASH2x (validation set). The *offline validation* of the models was carried extensively on the Intel processor for a proof-of-concept using the SPEC CPU 2006 benchmark suite (Section 4.1.2). Nevertheless, the models built have shown very high accuracy in predicting performance, and power over three different architectures (as will be shown in Section 4.2).

We have also conducted a study with varying number of training benchmarks with all possible combinations, i[that is] two floating point, and two integer, one floating point, three integer, etc., and the prediction error was not acceptable [63–65].

To build models offline, we collect traces which contain the PMCs samples of individual components per application. The traces, obtained at different P-States, need to be realigned

Table 4.3 Formula to compute activity ratios. Events and *perf* raw event numbers used on AMD (top), ARM (middle) and Intel (bottom) machines

Component	AMD	AMD (*perf* raw event)
FE	RETIRED_UOPS	R5000C1:U
INT	DISPATCH_STALL_FOR_INT_SCHED_QUEUE_FULL	R0D6
FP	RETIRED_MMX_AND_FP_INSTRUCTIONS:X87	R5001CB:U
BPU	BRANCH_RETIRED	R5000C3:U
L1	PERF_COUNT_HW_CACHE_L1D	-
L2	REQUESTS_TO_L2	R037D
L3	PERF_COUNT_HW_CACHE_REFERENCES	R080
MEM	PERF_COUNT_HW_BUS_CYCLES	R0062

Component	ARM	ARM (*perf* raw event)
FE	-	-
INT	INST_SPEC_EXEC_SIMD	R074
FP	INST_SPEC_EXEC_VFP	R075
BPU	INST_SPEC_EXEC_SOFT_PC	BRANCHES
L1	L1D_CACHE	R04
L2	L2D_CACHE	R16
L3	NO L3	NO L3
MEM	PERF_COUNT_HW_BUS_CYCLES	BUS-CYCLES

Component	Intel	Intel (*perf* raw event)
FE	UOPS_RETIRED:ANY	R1C2
INT	(UOPS_DISPATCHED_PORT:(PORT_0 + PORT_1 + PORT_5) - FP_COMP_OPS_EXE:X87 - BR_INST_RETIRED)	R1A1 + R2A1 + R80A1 - R110 - R0C4
FP	FP_COMP_OPS_EXE:X87	R110
BPU	BR_INST_EXECUTED	RFF88
L1	PERF_COUNT_HW_CACHE_L1D	-
L2	L2_RQSTS:0XC0	RC024
L3	LAST_LEVEL_CACHE_REFERENCES	R4F2E
MEM	PERF_COUNT_HW_BUS_CYCLES	BUS-CYCLES

in order to compare the activity ratios at similar points of execution (Section 4.1.1.5). This realigning is done with respect to the total instruction and exclude the trace points for which the difference in total instructions retired between two trace points for a given application exceeds ten million instructions. This process of realigning is to ensure that the trace points are at the same point of execution, and makes the activity ratio comparable.

We empirically select a sampling period of 250 ms to collect PMCs and power using the RAPL register for building offline models as it increases the total number of trace points for a given application without exceeding ten million instructions.

4.1.1.2 Modelling Power

Power can be modelled based on the PMCs [63–65]. Multilinear regression models allow for power prediction with high accuracy (less than 5% error rate). To the best of our knowledge, all the models previously proposed only predict the power consumption in the current hardware configuration.

Predicting power in the current configuration — We build on the technique proposed by Bertran et al. [66] to predict power at the current configuration (ρ_c), and is computed based on the activity recorded in FP, FE, BPU, L1 cache, L2 cache, LLC cache and MEM.

$$\rho_c = \sum_{i=0}^{comps} (\Delta_i \times AR_i) + constant \tag{4.1}$$

Equation 4.1 represents the multi linear regression model for predicting ρ_c, where Δ_i and *constant* represents the coefficients to be learned and AR_i represents the activity ratio of the individual components. We build one model for each available P-State with Cl-State zero.

Predicting power across P-States — Intuitively, one of the biggest challenges to predict performance or power across P-States, while keeping the Cl-States fixed, is the need to interpolate the component activity ratios with the higher or lower P-State. Contrary to our intuition, our findings show that the changes in activity ratio of the microarchitectural components are minimal.

For example, Table 4.4 shows the average activity ratio at the minimum and maximum P-State for the microarchitectural components INT, LLC, and FP for 16 SPEC benchmarks [37] for the first 100 million instructions of execution. Specifically, *Lb*change nory intensive floating point benchmark) shows a 67.4% change in activity ratio from 0.8 GHz to 2.4 GHz in FP. However, observe that the absolute change in activity ratio is negligible.

We visually represent an example from Table 4.4 in Figure 4.1. The Figure shows the average activity ratio of the microarchitectural components for the first 100 million instructions of execution across all P-States. Specifically, Figure 4.1a shows a 65% change in activity ratio from 0.8 GHz to 2.4 GHz in FP. However, observe that the absolute change in activity ratio is negligible. To ensure the statistical significance of this finding, we ran the workloads multiple times and had a 95% confidence with a low error margin (less than 2%).

Table 4.4 Activity range for SPEC CPU 2006. The activity computed over 100 million instructions at minimum and maximum P–States

Benchmark	INT/FP	INT	LLC ($\times 10^{-2}$)	**FP** ($\times 10^{-1}$)
Xalancbmk	INT	0.6835-0.5124	2.1426-1.7193	0.0021-0.0012
Calculix	FP	1.7852-1.8618	0.2043-0.2074	0.0139-0.0127
GemsFDTD	FP	1.3728-1.2165	2.9210-2.6255	0.0156-0.0168
Wrf	FP	1.0714-1.0526	0.3263-0.2859	0.1673-0.0941
Tonto	FP	1.2196-1.2258	0.1295-0.4179	0.0957-0.2169
Povray	FP	1.0132-1.0290	0.0808-0.0764	0.4823-0.4958

Benchmark	INT/FP	INT	LLC ($\times 10^{-2}$)	**FP** ($\times 10^{-5}$)
CactusADM	FP	1.0972-0.8945	0.5738-0.4455	1.6002-0.5251
Lbm	FP	1.0231-1.0199	1.1000-1.7063	1.6121-0.5255
Hmmer	INT	1.3572-1.3945	0.1099-0.0862	3.1287-1.6042
H264ref	INT	1.2756-1.3068	0.2752-0.2565	0.0344-0.0353
Gobmk	INT	0.6718-0.6745	0.3105-0.2924	0.0502-0.0446
Bzip2	INT	0.9543-0.9525	0.9027-0.9270	1.6428-0.5573
Astar	INT	0.8335-0.7726	2.1482-1.5707	1.8260-0.7140
Mcf	INT	0.3398-0.2619	2.6239-2.8608	1.6472-0.5575

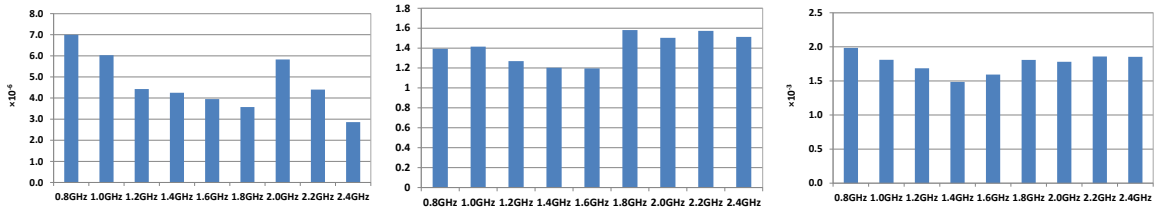
This behaviour is consistently observed across any benchmark for any microarchitectural component, as it is an microarchitectural property; An analogy for this can be drawn to a unique finger print for every person.

On the other hand, having negligible absolute difference in activity ratio across P-States does not imply that the power across P-States for a given Cl-State is constant, but instead the difference in the real activity ratio across P-State is reflected using the coefficients derived in Equation 4.1.

Therefore, the PMCs read at the current configuration can be used to predict power or performance at P_f while the Cl-State remains constant.

Predicting power across Cl-States — To predict power (ρ_f) across Cl-States, moving from Cl_c to Cl_f, we build a multilinear regression model based on the ratio of the Cl-States and predicted power at the current P-State (ρ_c).

$$\rho_f = (\Delta \times \rho_c + (\beta \times \frac{Cl_f}{Cl_c})) + constant \tag{4.2}$$



(a) Floating Point Unit **(b)** Integer Unit **(c)** LLC Cache

Figure 4.1 Component activity ratio. The activity measured over 100 million instructions at all P–States for *lbm* from the SPEC CPU 2006 benchmark suite.

Equation 4.2 represents the multi linear regression model for predicting ρ_f, where Δ, β and *constant* represent the multi linear regression coefficients and we build one model at each P-State.

4.1.1.3 Modelling Performance

Modelling performance is based on three major constraints which are: (1) the inherent instruction-level parallelism of the execution flow; (2) number of stall cycles caused by misses in the cache hierarchy; (3) the number and latency of functional units such as INT and FP. The relationship between these constraints is very difficult to predict. Therefore, the most accurate way to estimate performance is by implementing a full-scale simulator [67, 68]. However, this increases the complexity of predicting performance in runtime and suffers a higher latency. On the other hand, linear regression techniques allow faster predictions and are easier to implement although they suffer from a relatively high error rate compared with simulation techniques.

Reporting performance at current configuration — The PMCs available in the current hardware subsystems allow precise measurement of the number of instructions retired (`INST_RETIRED`). We take advantage of this counter and report the number of instructions retired (in millions) per second (MIPS), that is, η_c.

Predicting performance across P-States — We predict the performance in MIPS (η_f) of a thread using multi linear regression models. The components modelled are MIPS, instructions per cycle (IPC), private L1, private L2 and LLC cache and the ratio of change of P-State moving from P_c to P_f.

$$\eta_f = \sum_{i=0}^{comps} (\Delta_i \times AR_i) + (\beta \times \frac{P_f}{P_c}) + constant \tag{4.3}$$

Equation 4.3 represents the multi linear regression model for predicting η_f, where Δ_i, β and *constant* represent the coefficients to be learned and AR_i represents the activity ratio of the individual components. We build one model for each available P-State with no idle cycles.

Predicting performance across Cl-States — Similarly, for predicting performance (η_f) across Cl-States, moving from Cl_c to Cl_f, we build multi linear regression models while keeping the P-State fixed and moving across the Cl-States. The components modelled are MIPS, IPC, private L1, private L2 and LLC cache and the ratio of change of Cl-State.

$$\eta_f = \sum_{i=0}^{comps} (\Delta_i \times AR_i) + (\beta \times \frac{Cl_f}{Cl_c}) + constant \tag{4.4}$$

Equation 4.4 represents the multilinear regression model for predicting η_f, where Δ_i, β and *constant* represents the coefficients to be modelled and AR_i represents the activity ratio of the individual components. We build one model for every P-State available.

4.1.1.4 Single-core algorithm

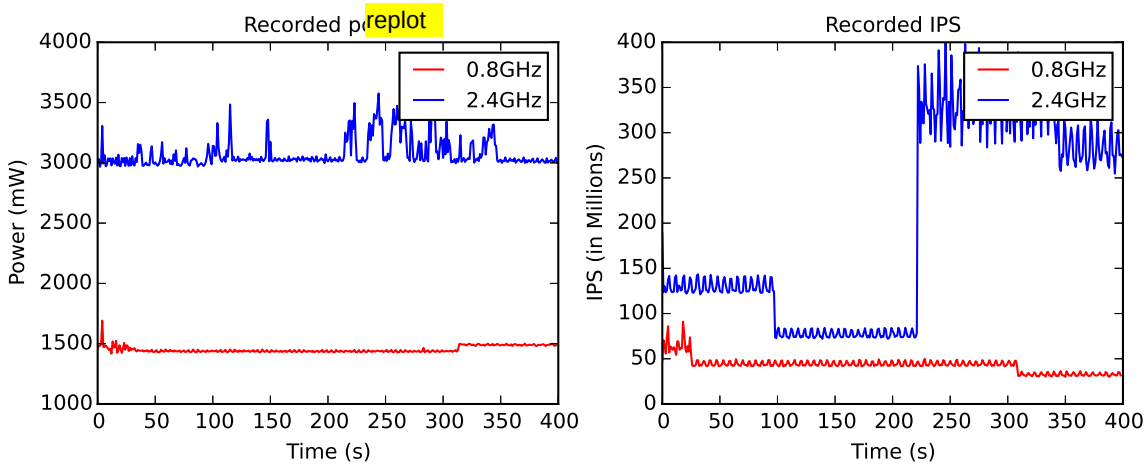
The single core algorithm predicts performance and power for a given P-State and Cl-State in a single thread using multilinear regression models.

1. Read the PMCs at current configuration and compute the activity ratios.
2. Predict power and report performance at current configuration.
3. Predict power and performance for current Cl-State and future P-State.
4. Predict power and performance for current P-state and future Cl-State.

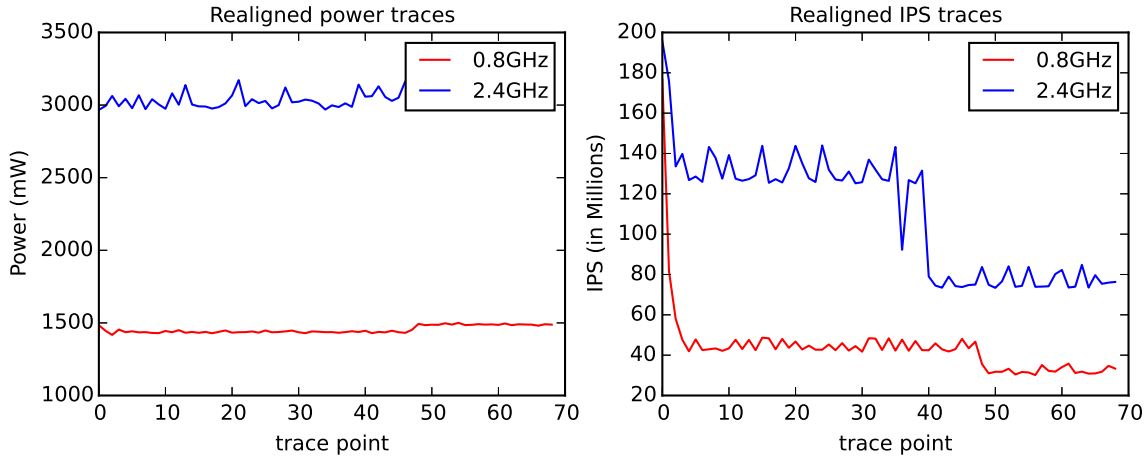
4.1.1.5 Trace files

The offline models are built using the statistical information gathered, and stored in trace-files. The trace-files contain activity ratio for individual components for each benchmark. The traces, obtained at different P-States, Cl-States need to be realigned such that the activity ratios are comparable at similar points of execution.

The performance statistics for each benchmark in the training dataset are gathered by binding the benchmark to the cores using the Linux system call `sched_setaffinity`. Assigning the affinity [69] for a thread bypasses the completely fair scheduler (CFS) and helps avoid excessive thread migration and provides a more controlled environment.



(a) Power and MIPS with variation in time.



(b) Power and MIPS traces realigned with IPS.

Figure 4.2 Traces to build models. From left-to-right, power, and performance *recorded* using the power meter, and PMCs (on top), and traces *realigned* based on IPS (on bottom) for the SPEC benchmark astar.

Realigining traces gathered at P-State, Cl-State to build models offline provides a unique challenge because the correlation between MIPS, the frequency of execution (P-State), and the time are non-linear.

For instance, Figure 4.2a shows the time varying demands for performance and power consumed (in mW) for the SPEC benchmark *astar* executing at 0.8 GHz and at 2.4 GHz with Cl-State zero. As can be seen, the totally number of retired instructions at 0.8 GHz and 2.4 GHz are not equal. Observe that the total performance for the first 300 seconds at 0.8 GHz and first 100 seconds at 2.4 GHz is the approximately same. This makes it clear that the correlation between frequency, and total performance is not one-to-one. Therefore,

a prediction model based on time can not provide a measure of performance or power at similar point in the execution of the thread and thus, raises the need for realigning traces with respect to total performance.

As indicated before, we use a buffer of ten millions instructions to compare traces for each benchmark, to ensure that the boundary of error is fixed (!). Figure 4.2b shows the traces realigned with respect to total performance for *astar* at 0.8 GHz and 2.4 GHz. Each trace gathered refers to approximately similar points of execution in the thread. This methodology is followed for every P-State, and Cl-State combination, and such traces here-forth are referred to as *realigned traces*.

emph instead of it

4.1.2 Evaluation

In this section, we evaluate the models to predict both performance, and power using the SPEC CPU 2006 benchmark suite for the traces gathered. Next, we evaluate the single-core models when predicting performance, and power at runtime/online. Both the evaluations are carried out for numerous P-State and Cl-State combinations.

4.1.2.1 Algorithm Configuration

nishtala: write about mapping interval, information required to map

offline modelling is run only once ebcause it is OFFLINE!. runtime stuff is done three times with error less than 2%

4.1.2.2 Model Assessment

The evaluation of the prediction models in the remainder of this chapter are evaluated, and shown in terms of percentage absolute average error (PAAE), and the standard deviation (STDEV) for each data point over a period of 300 seconds:

$$PAAE = \frac{1}{N}\left(\sum_{i=1}^{N}\frac{|M_i - m_i|}{m_i}\right) \tag{4.5}$$

Where N is the number of data points for each benchmark, M is the predicted value, and m is the measured value (Reproduced from [70]).

The PAAE is a well-known metric [66, 71, 72, 70] to evaluate the predicted values used to meet the performance (or power) requirements and the actual values reported by PMCs for performance (or by power monitors for power consumption). The standard deviation over PAAE determines how far the prediction is from the optimal point.

4.1.2.3 Single-Core offline evaluation

We performed evaluation for a subset of P-States, and Cl-States. The P-States combinations we evaluate are 0.8 GHz, 1.6 GHz, 2.4 GHz; similarly, we consider evaluation at 0%, 10%, 30%, and 50% variation in sleep cycles.

In the single-core offline validation technique, the PAAE is computed using the error between the PMCs gathered to predict power, and performance, and the value recorded using the PMCs at the corresponding number of instructions retired. This information is stored in the realigned trace file. The PAAE, and STDEV are computed over a period of 300 seconds.

Table 4.5 shows the power prediction error when switching between P-States across current P-State (P_c) to future P-State (P_f). Our results have shown that the PAAE over the subset of switches in P-States is less than 3% (mean), while the maximum error is 5.75%.

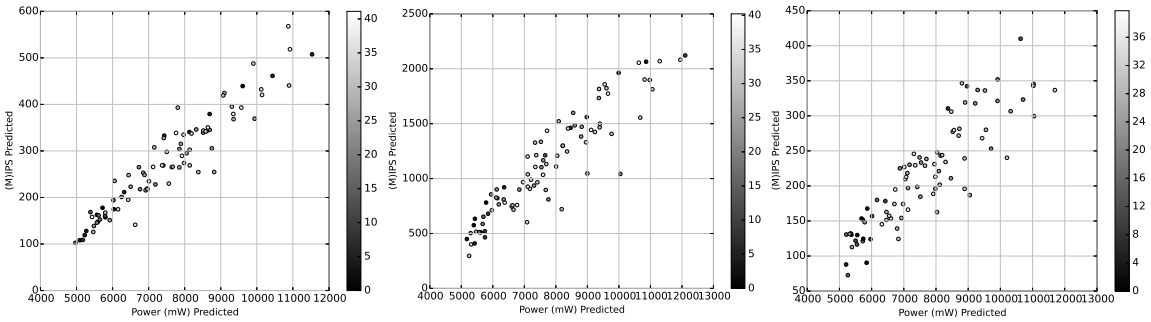
Table 4.6 details the performance prediction error when switching between P-States across current P-State (P_c) to future P-State (P_f). The worst-case scenario PAAE is 16.3% for the memory-intensive benchmark *mcf*. Also note that there exists a collinearity between PAAE, and the ratio of change in P-State. For instance, observe there is linear rise (or decrease) in PAAE for benchmark *Astar* when switching from current P-State to a higher P-State (or lower).

Table 4.7 shows the power prediction error when switching between Cl-States across current Cl-State (Cl_c) to future Cl-State (Cl_f) at 1.8 GHz. Our results have shown that the PAAE over the subset of switches in Cl-State is less than 5% (mean), with a maximum error of 10%.

Table 4.8 shows the performance prediction error when switching between Cl-States across current Cl-State (Cl_c) to future Cl-State (Cl_f) at 1.8 GHz. The maximum error observed when switching between the Cl-States is 15.7% (with maximum error of 22.43% for *GemsFDTD*).

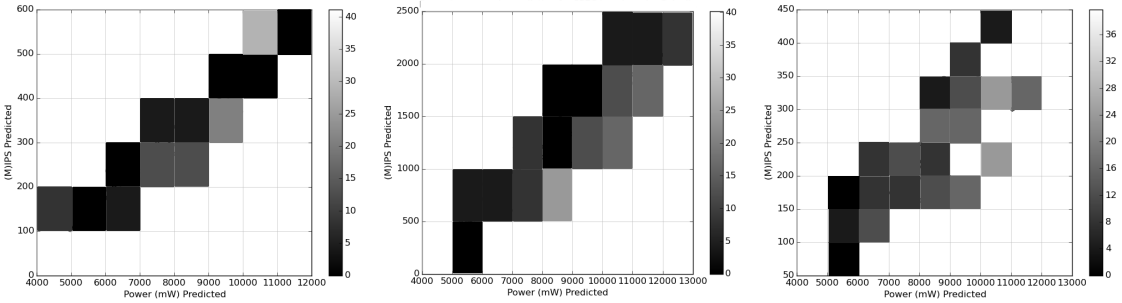
The results obtained so far demonstrate that the error across numerous switches, for both P-States, and Cl-States, in an offline modelling technique is "acceptable" [62, 60, 49, 61, 72, 71], and determine the real behaviour of the application.

We demonstrate the effectiveness of the technique using three benchmarks from the SPEC suite with ranging from compute-intensive to memory-intensive. Figure 4.3 represents the predicted performance (y-axis) and power (x-axis) for *astar* (mid memory intensive), *mcf* (memory bound) and *Calculix* (compute bound) at nine P-States and six Cl-States. The initial configuration is set to 0.8 GHz and Cl-State zero. Each point in the graph represents a unique hardware configuration. The colour-map indicates the maximum PAAE between performance, and power (max_{PAAE} = max($PAAE_{power}$, $PAAE_{performance}$)). Observe that there exist multiple points grading from *black* (low PAAE) to white (higher PAAE) distributed



(a) astar **(b)** calculix **(c)** mcf

Figure 4.3 Percentage prediction error for SPEC benchmarks. The power and performance prediction error for Astar (mid memory intensive), Calculix (compute intensive), and Mcf (memory intensive).



(a) astar **(b)** calculix **(c)** mcf

Figure 4.4 Percentage prediction error per grid for SPEC benchmarks. The power and performance prediction error for Astar (mid memory intensive), Calculix (compute intensive), and Mcf (memory intensive).

across the grid. Figure 4.4 represents the data plotted in Figure 4.3 in terms of maximum error per sub-grid, where a configuration exists. Figures 4.3 and 4.4 show that there exists at least one prediction with less than 15% error. This behaviour is observed across all SPEC benchmarks.

Table 4.5 Power error across P-States. Error shown in terms of PAAE, and STDEV

Benchmarks	Lower - Higher						Higher - Lower					
	0.8-1.0		0.8-1.6		0.8-2.4		1.0-0.8		1.6-0.8		2.4-0.8	
	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV
astar	1.772	1.282	1.492	0.971	1.172	0.779	1.735	0.670	1.718	0.713	2.245	2.580
bzip2	1.927	0.945	1.922	2.482	1.401	0.981	1.136	0.679	0.824	0.525	2.053	1.727
calculix	1.267	1.195	1.274	1.133	3.480	2.098	1.114	0.572	2.371	2.245	2.356	2.042
gemsFDTD	3.882	2.773	3.541	2.127	4.238	2.303	3.739	2.952	3.781	2.241	3.357	1.704
gobmk	1.054	0.940	1.339	1.605	2.409	3.201	0.856	0.652	0.697	0.466	0.665	0.572
hmmer	2.600	0.528	2.523	0.861	3.062	1.452	2.738	0.618	2.696	0.598	2.307	0.927
lbm	0.849	0.686	1.824	1.101	2.155	1.380	0.840	0.697	1.431	0.825	1.473	0.771
leslie3d	1.095	0.862	1.324	1.283	2.036	1.200	1.240	0.958	1.165	0.931	1.377	1.233
libquantum	2.676	3.906	1.643	1.255	2.435	1.953	3.330	2.766	1.957	1.496	1.852	1.852
mcf	3.172	2.355	3.785	2.677	5.666	2.249	2.419	2.656	2.292	2.227	2.422	2.724
povray	1.143	0.615	2.049	1.093	1.897	3.250	1.234	0.514	1.396	0.546	1.060	0.466
soplex	5.132	1.722	2.761	1.629	2.816	1.163	5.775	1.988	5.687	1.880	4.135	1.706
xalancbmk	2.522	1.587	3.627	1.580	5.807	1.457	1.986	1.358	1.617	1.154	1.401	1.242
Avg.	2.238	1.492	2.239	1.523	2.967	1.805	2.165	1.314	2.126	1.219	2.054	1.504

Table 4.6 Performance error across P-States. Error shown in terms of PAAE, and STDEV

Benchmarks	Lower - Higher						Higher - Lower					
	0.8-1.0		0.8-1.6		0.8-2.4		1.0-0.8		1.6-0.8		2.4-0.8	
	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV
astar	9.409	16.745	11.174	14.054	24.076	9.518	7.923	8.647	9.126	9.41	16.242	14.443
bzip2	13.163	11.362	17.277	11.752	22.979	15.542	13.061	10.495	19.802	17.591	22.663	13.478
calculix	22.19	13.4358	19.24	15.115	26.968	20.412	17.219	26.997	18.165	24.183	19.888	23.932
gemsFDTD	21.687	15.48	23.884	14.5	15.65	14.648	21.707	15.58	22.433	12.316	12.709	9.618
gobmk	8.968	7.824	8.742	8.686	9.236	7.158	8.852	7.607	8.889	9.119	9.614	8.045
h264ref	6.095	4.689	9.177	14.613	1.887	1.353	6.116	4.707	7.705	8.513	1.94	1.42
hmmer	5.704	5.982	6.088	5.342	6.999	4.742	5.985	7.567	6.136	5.337	7.434	5.26
lbm	6.931	5.103	5.824	4.122	5.773	3.866	6.843	4.984	5.734	4.009	5.629	3.771
libquantum	8.602	6.359	10.3	7.095	12.577	10.284	8.635	6.521	9.545	6.445	10.541	7.697
mcf	11.081	9.698	17.721	15.106	29.327	20.954	10.441	9.11	14.672	12.777	21.431	11.328
povray	6.15	4.415	12.09	22.553	6.932	5.114	6.241	4.673	9.145	10.637	7.121	5.42
soplex	8.829	6.744	10.344	8.555	16.778	13.797	8.192	5.758	9.039	6.553	13.366	8.671
xalancbmk	9.653	8.565	19.31	13.979	33.991	16.745	9.702	9.427	15.461	9.203	25.172	11.007
Avg.	10.651	8.954	13.167	11.959	16.398	11.087	10.071	9.390	11.989	10.469	13.365	9.545

Table 4.7 Power error when switching across Cl-States at 1.8GHz. Error shown in terms of PAAE, and STDEV

Benchmarks	Lower - Higher						Higher - Lower					
	0-10		0-30		0-50		10-0		30-0		50-0	
	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV
astar	3.190	2.308	2.686	1.748	2.110	1.402	3.123	1.206	3.092	1.283	4.041	4.644
bzip2	3.469	1.701	3.460	4.468	2.522	1.766	2.045	1.222	1.483	0.945	3.695	3.109
calculix	2.281	2.151	2.293	2.039	6.264	3.776	2.005	1.030	4.268	4.041	4.241	3.676
gemsFDTD	6.988	4.991	6.374	3.829	7.628	4.145	6.730	5.314	6.806	4.034	6.043	3.067
gobmk	1.897	1.692	2.410	2.889	4.336	5.762	1.541	1.174	1.255	0.839	1.197	1.030
hmmer	4.680	0.950	4.541	1.550	5.512	2.614	4.928	1.112	4.853	1.076	4.153	1.669
lbm	1.528	1.235	3.283	1.982	3.879	2.484	1.512	1.255	2.576	1.485	2.651	1.388
leslie3d	1.971	1.552	2.383	2.309	3.665	2.160	2.232	1.724	2.097	1.676	2.479	2.219
libquantum	4.817	7.031	2.957	2.259	4.383	3.515	5.994	4.979	3.523	2.693	3.334	3.334
mcf	5.710	4.239	6.813	4.819	10.199	4.046	4.354	4.781	4.126	4.009	4.360	4.903
povray	2.057	1.107	3.688	1.967	3.415	5.850	2.221	0.925	2.513	0.983	1.908	0.839
soplex	9.238	3.100	4.970	2.932	5.069	2.093	10.395	3.578	10.237	3.384	7.443	3.071
xalancbmk	4.540	2.857	6.529	2.844	10.453	2.623	3.575	2.444	2.911	2.077	2.522	2.236
Avg.	4.028	2.686	4.030	2.741	5.341	3.249	3.897	2.365	3.826	2.194	3.697	2.707

Table 4.8 Performance error when switching across Cl-States at 1.8GHz. Error shown in terms of PAAE, and STDEV

Benchmarks	Lower - Higher						Higher - Lower					
	0-10		0-30		0-50		10-0		30-0		50-0	
	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV	PAAE	STDEV
astar	9.836	9.799	12.440	8.439	23.711	11.462	9.394	7.750	11.462	8.602	13.588	11.935
bzip2	12.486	8.747	19.400	12.959	28.110	16.538	12.847	10.342	19.678	11.790	23.965	19.870
calculix	23.681	12.9840	17.184	16.378	37.767	22.141	17.679	14.590	19.893	8.623	23.289	6.008
gemsFDTD	17.164	14.747	15.109	9.980	24.321	16.465	16.932	15.113	15.312	11.467	23.785	17.364
gobmk	9.290	5.734	12.157	8.156	24.456	11.360	9.499	5.947	11.563	7.668	14.496	10.778
h264ref	10.302	5.408	10.646	10.863	25.193	12.028	10.669	6.161	12.767	8.660	15.129	9.913
hmmer	7.408	4.300	10.465	6.956	24.085	12.732	7.382	4.372	8.186	6.064	16.058	10.183
lbm	10.238	5.787	12.015	7.584	25.261	12.579	10.339	6.474	11.138	7.127	15.771	13.232
libquantum	8.857	6.154	12.238	10.852	24.808	12.907	8.865	5.854	13.729	12.958	14.651	10.185
mcf	9.689	7.396	15.263	10.717	27.495	13.709	10.650	9.180	14.185	18.806	19.416	21.161
povray	8.500	5.055	9.751	8.519	23.424	10.504	8.718	5.406	11.298	8.313	12.902	6.855
soplex	7.855	5.257	13.407	8.983	25.487	14.176	7.972	5.204	12.413	7.405	17.496	16.825
xalancbmk	9.528	7.612	10.698	9.382	31.952	15.761	9.000	6.764	11.377	11.185	14.587	8.897
Avg.	11.141	7.614	13.136	9.982	26.621	14.028	10.765	7.935	13.308	9.898	17.318	12.554

4.1.2.4 Single core online evaluation

The evaluation of the results for a single core model are described in two steps. First, we evaluate, and summarise the results for over 150 applications using `K-Means` [73]. Then, we expand the results obtained for each benchmark per benchmark suite. Irrespective of the form in which the results are presented – we predict performance, and power across a combinations of P-State, and Cl-State at runtime. The PAAE on a single core is computed using the error between value measured from PMCs for performance (and power meters for power) and predicted performance or power, and the error bars represent the STDEV.

First, we separate the benchmarks used in validation into four `clusters`, to present the results for over 150 single-threaded applications, using `K-Means` with parameters FE, INT, FP, MEM, BPU, L1, L2, L3. The number of clusters (four) was chosen empirically based on the silhouette coefficient. We narrow the number of parameters to two using principal component analysis for keeping the most singular vectors to project the data in a lower dimensional space. Clusters are named with the architecture and cluster number, such as ARM-0, Intel-3, etc. Each cluster has results from all four categories: Insensitive, Cache-Friendly, Cache-Sensitive, and Thrashing.

Figure 4.5 shows the average PAAE over all applications in a cluster on a single core, and Figure 4.7 presents the results for each application in detail on a single core. The results in the Figure 4.7 are organised as follows: Intel (top row (a)), AMD (middle row (b)), and ARM (bottom row (c)). We analyse the data points with the higher error and also pointed out the sources of error below.

(a) Average PAAE when predicting performance for Intel-2 for *thrashing* benchmarks is 15.8% because *Mcf* has 22.5% error as it is a pointer-chasing benchmark [37] and generates more than 41000 LLC misses per million instructions retired and the models are not trained

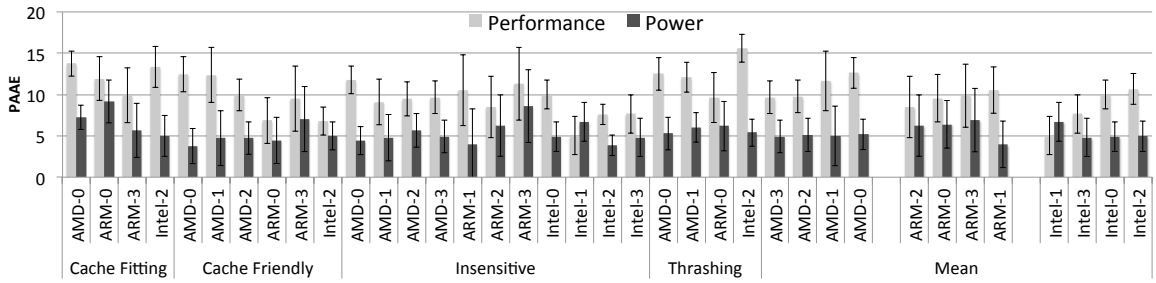


Figure 4.5 Single core online validation. Average PAAE when predicting performance and power on a single core for across a combination of P-States, Cl-States and architectures. The error bars represent the STDEV

.

for that range. On the other hand, applications like *Lbm – memory intensive – floating point benchmark* generate 3000 LLC misses per million instruction retired has an error 11.2%

(b) The average PAAE for performance for *Cache-Friendly* on AMD-0 and AMD-1 are 12.4% and 12.3%, respectively; this is because both clusters contain applications such as *Canneal* and *dedup*. The possible sources of error are: (1) Both applications have a high dynamic variability in application phases [74], which leads to erroneous counters due to PMCs multiplexing. (2) In contrast to the other applications across suites, these benchmarks have a shorter execution time. (3) Observe that Canneal is a cache fitting benchmark on Intel, by contrast it is a cache-friendly benchmark on AMD. This is because of the aggressive hardware prefetcher on Intel causing a higher miss rate [75], thereby leading to fewer dynamic phase changes and relatively smaller error of 6.5%.

We also observe that application `radix` is a cache-fitting, integer radix sorting algorithm, has very high activity in FE, across three different architectures, even though other benchmarks across four suites do not show this behaviour.

Figure 4.6 shows average PAAE over all applications in each benchmark suite across architectures, with error bars representing STDEV. Across architectures, we observe performance PAAE is higher for SPEC benchmarks, which have high variability at runtime, and low for NAS benchmarks, which have less variability after the initialisation phase. We conclude that the models to predict performance and power are accurate enough to capture the real behaviour, and since the computational complexity at runtime is low, they can be used for fine-grain power or performance management. The models to predict power can be built using standardised power meters and the models are built using PMCs that are available across architectures.

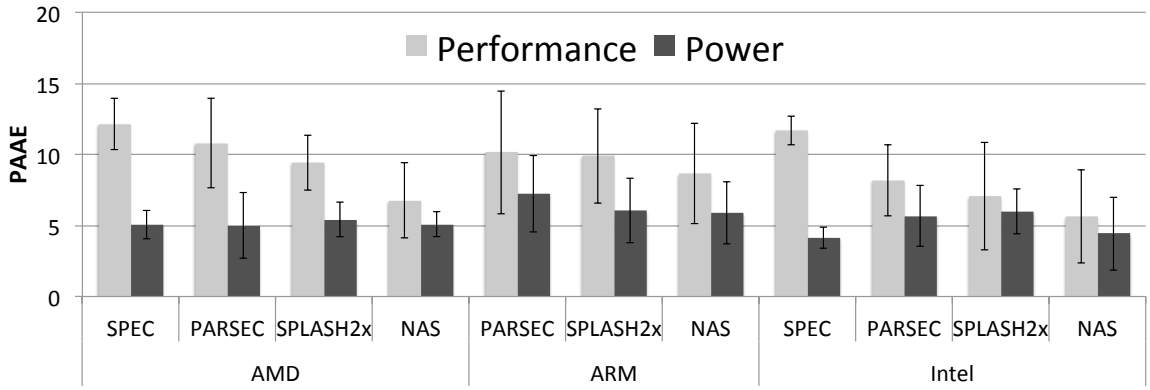
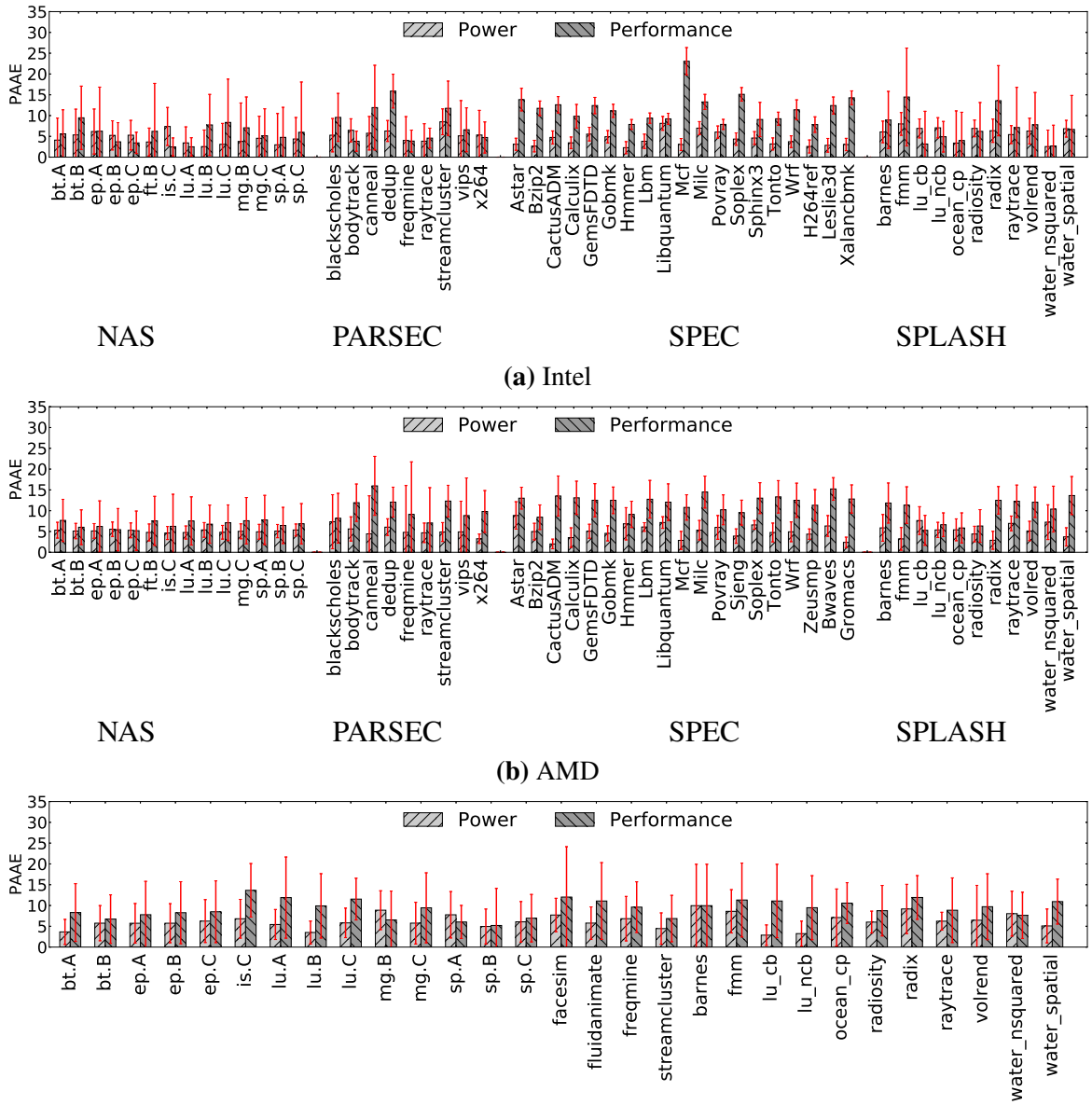


Figure 4.6 Single core online validation per suite. Average PAAE when predicting performance and power per suite across architectures. The error bars represent the STDEV.



(a) Intel

(b) AMD

(c) ARM

Figure 4.7 Single core online validation per benchmark. Average PAAE when predicting performance and power on a single core for each benchmark from four benchmark suites across a combination of P-States, Cl-States and architectures. The error bars represent the STDEV. Results are shown for Intel (top), AMD (middle), and ARM (bottom). The benchmarks are organised as follows from left-to-right: NAS, PARSEC, SPEC, and SPLASH2x

.

In addition, it is observed that when predicting performance at the future P-State on Intel processors, the error varies depending on the size of the leap between current P-State to future P-State or current Cl-State to future Cl-State. For every switch in P-State, we calculate

Table 4.9 Error for combinations of P-States and Cl-States using REPP. PAAE for every combination of switch in P-State, and a specific set of differences when switching between Cl-States on a single core Intel architecture. Similar results were observed across AMD, and ARM.

Leap in P-State	1	2	3	4	5	6	7	8	-
P-State	14.14	16.75	23.37	15.37	13.78	57.17	18.49	27.49	-
Leap in Cl-State	5	10	15	20	25	30	35	40	45
Cl-State	24.86	26.25	29.14	26.36	23.25	22.07	29.30	31.94	34.56

the PAAE when switching between every combination of P-States. Similarly, we calculate the PAAE for Cl-States when the difference in switches is a multiple of five for every P-State. As can be seen in Table 4.9, with a higher switch in hardware configurations from the current configuration, a greater error is observed. This is because, we train the models using a small subset of benchmarks, and use it for a wide range of threads which are not a part of the training set. Similar results were observed across ARM, and AMD.

4.1.3 Conclusion

Our single-core models for the prediction technique are built such that the coefficients are not dependent on the number of applications. Instead they are based on the activity in each of the microarchitectural components. Therefore, it can scale across multi-node, multi-core data centres, and is aimed to ensure user-defined constraints for power, and performance are met. In contrast to prior work [76], REPP is not based on application signature or similarities between application. Instead, REPP leverages online monitoring of basic PMCs that provide application behaviour at runtime, which require a one-time, offline profiling effort per architecture, and not per application. Such data is then fed into statistical tools to predict performance and power, making REPP fast, accurate, and architecture-agnostic (see Table 3.3).

However, such single core models are built, it is important to understand that modern data centres are equipped with multiple nodes consisting of racks of multicore servers, instead of a single core, thus making the existence of REPP alone futile. In the next section, we introduce REPP-H, which in contrast to REPP, is a modelling and prediction technique for both single-threaded, and multiprogrammed workloads in data centres.

4.2 REPP-H: Runtime Estimation for Performance and Power in Heterogeneous Data Centres

In the previous section, we introduced offline procedure to build performance, and power models, and also the offline, and online validation of the single-core models on three different architectures. In this section, we extend the model built to predict for single-threaded and multi-programmed in data centres by first tackling the issue of shared resource contention, and then use the predictions made to control power, and performance in multicore servers running multiprogrammed workloads.

4.2.1 Multi-core modelling

To predict total performance or power in multi-core architectures, we *aggregate* the results from each of the *single core* models (Section 4.1.1.4).

Single core models when exposed to multi-core prediction techniques are bound to suffer from an error due to the contention for shared resources [57, 49, 77–84]. To compensate for this contention, we model the error in performance and power using four different types of workloads with variations in memory footprint and validate the models, by switching between combinations of P-States and Cl-States.

It impossible to be at two P-States or Cl-States at the same time. Therefore for training and validating the models, we generate multiple random tuples of P-State, Cl-State combinations per core within the minimum and maximum P-State and Cl-State ranges.

The estimated increase in power consumption and degradation in performance due to the shared resource contention is modelled offline using linear regression techniques. We build one model for performance and another model for power. We demonstrate the process of building models for one core. This process is replicated across all cores simultaneously.

1. Read the 1000 random tuples of P-State and Cl-State per core.
2. Set current configuration for core to the current tuple and future configuration for the same core to next tuples.
3. Spawn training workloads consecutively.
4. Apply the single core algorithm (Section 4.1.1.4) for the thread running on the core to predict power and performance in the future configuration.
5. Switch to future configuration after 250 ms.

(6) Report power and performance at future configuration using RAPL and PMCs respectively.

(7) Using the PMCs read at current configuration, compute activity ratios for private L1, private L2, LLC and MEM for the thread.

(8) Compute difference between the PMC read (or RAPL register) and prediction for performance (or power) using *REPP* per thread in the workload models the error due to shared resource contention.

The aforementioned method is followed for all random tuples generated for all cores for a period of 300 seconds. The sleep interval of 250 ms was chosen empirically as most SPEC benchmarks have less than 1% change in performance or power at a particular P-State and allowed us to predict in the same phase.

$$Error = \sum_{i=0}^{comps} (\Delta_i \times AR_i) + constant \tag{4.6}$$

Equation 4.6 represents the multi linear regression model to predict the error due to shared resource contention. Where Δ_i represents the coefficient to be learnt and AR_i represents the activity ratio of the individual components.

Intuitively, in a multi-core architecture as the number of threads increases, the performance per thread decreases and the total power consumption increases. Since the single-core models do not include contention in shared environments, the total power consumed will be lower and the performance higher. Therefore, the modelled error for power and performance is added and subtracted for every prediction on a per thread level respectively.

Then, we evaluate REPP-H across a wide range of performance and power constraints. Contrary to prior works [71, 68, 85, 86, 58], which predict power and performance at a system level, our work actually predicts at a system level on a per core basis.

With the shared resource contention model defined (Equation 4.6), the architecture of REPP for multi-core systems is complete. In the remainder of the sections, we summarise the architecture, a technique to select a configuration that can satisfy different power, and performance constraints.

4.2.1.1 REPP-H summary

nishtala: the single core model has a computation technique to predict at all configurations

Figure 4.8 shows a high-level view of REPP-H. REPP-H is a module that can be implemented across any multi-core server system running an operating system that allows

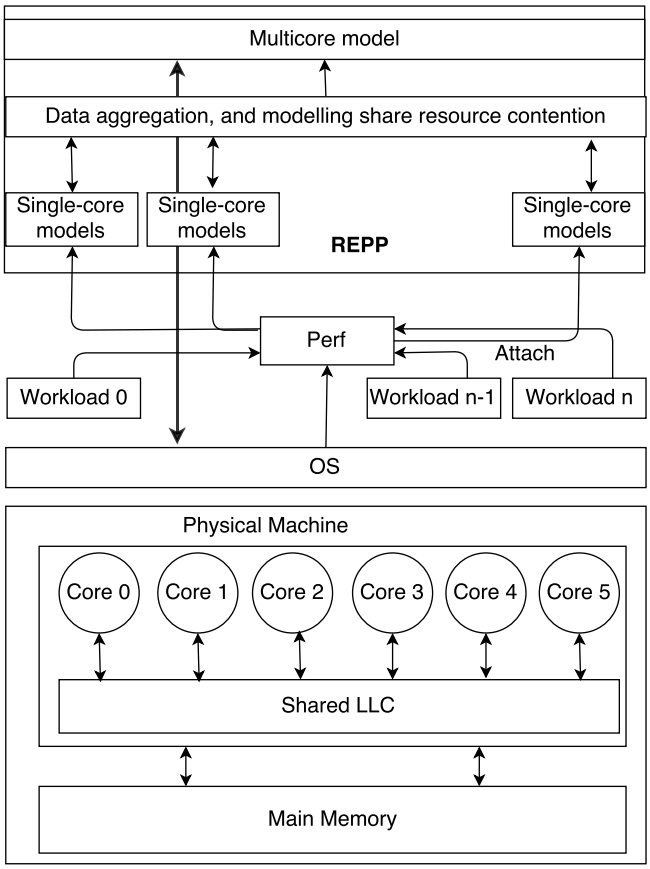


Figure 4.8 High-level view of REPP runtime system.

to gather performance statistics of workloads (e.g., `perf`, on all our multi-core machines). Such performance statistics are fed to REPP-H to compute the activity ratio's which are fed to the single-core models to compute the power, and performance across a wide range of P-States, and Cl-States. Since the single core models do not account for shared resource contention, Equation 4.6 models it. Then, the results from each of these single core models are *aggregated* to predict the power for a multi-core system. This results in the performance, and power modelling, and prediction technique across P-States, and Cl-States.

4.2.2 Multicore Evaluation

In evaluating REPP-H, we first show the modelling error when contention for shared resources is ignored. Next, we evaluate REPP-H when predicting performance, and power across a combination of P-States, and Cl-States on Intel architecture. The multiprogrammed workloads are generated using only benchmarks from SPEC, and PARSEC suites, based on the methodology described in section 3.4.1. Finally, we validate REPP-H by limiting the power usage, or delivering a minimum performance on all architectures using all the suites.

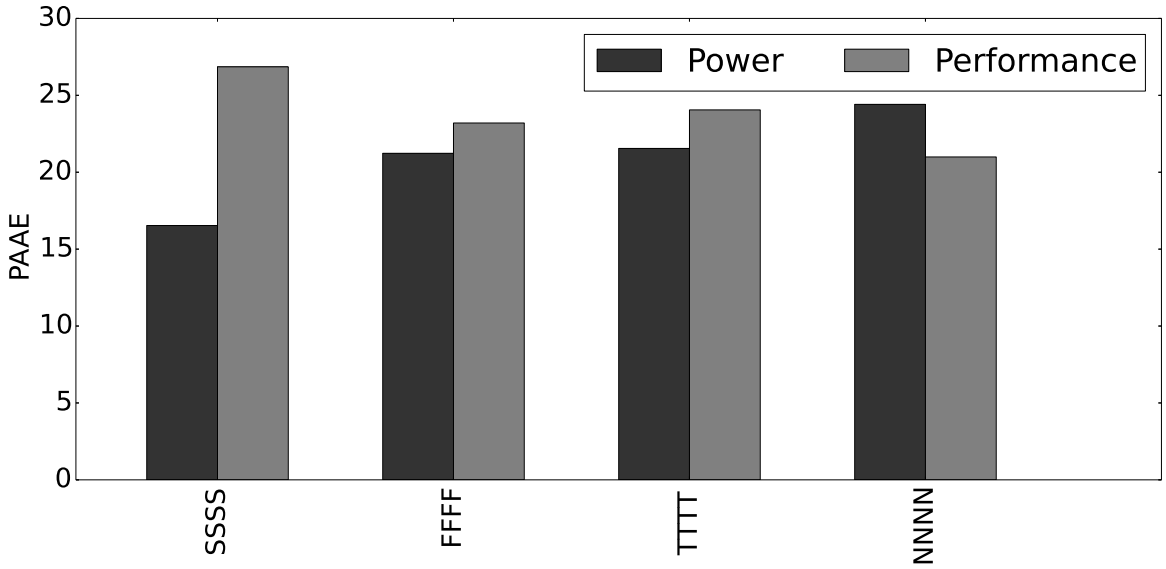


Figure 4.9 PAAE ignoring contention for shared resources. PAAE when predicting **power** and **performance** across 1000 combinations of P-States and Cl-States per core in multi-core architectures for training workloads

This process of delivering a minimum performance, or limiting the power usage is done by selecting a configuration from the predicted values that satisfies the constraint. Irrespective of how REPP-H is evaluated, the configuration of the algorithm is fixed.

4.2.2.1 Algorithm Configuration

nishtala: write about mapping interval, information required to map

Each experiment was run three times for each power and performance constraint. The deviation over multiple runs for each workload was low (<2%).

4.2.2.2 Evaluating multi-core models ignoring contention

Figure 4.9 shows PAAE in predicting power and performance across 1000 combinations of P-States and Cl-States per core in multi-core architecture. We highlight two key points from this graph. First, observe that workloads **SSSS** (a memory intensive workload) and **NNNN** (a compute intensive workload) have the highest PAAE when predicting performance and power, respectively, this is because the activity is predominant in the memory subsystem and processor, respectively. Second, observe that the error in predicting power increases as the compute intensiveness of the workloads increases, this is because the single core models aggregate the results obtained from each core and do not account for the contention due to shared resources, thereby the compute intensive workloads which have lower activity

in memory subsystem – compared to memory intensive workloads – are accounted for higher activity. Whereas the error when predicting performance increases as the memory boundedness of the workloads increase, this can be attributed to the fact that the activity generated per cycle in the components decreases, thereof the performance per thread. The PAAE and AAE when predicting power and performance across the training workloads are 20.93% (430 mW) and 23.8% (3316 MIPS).

fix this

4.2.2.3 Evaluating multi-core models including contention without constraints

In this section, we validate REPP-H by switching across all combinations of P-States and Cl-States on the Intel processor. Since the total number of P-State and Cl-State combinations are 41 billion on Intel processor (9 P-States, 50 Cl-States per core, and 4 cores) ref to the chapter of infr to validate for the entire spectrum. Therefore we generate 1000 random tuples of P-State, Cl-State combinations per core within the minimum and maximum P-State and Cl-State ranges. These random combinations are fixed across workloads.

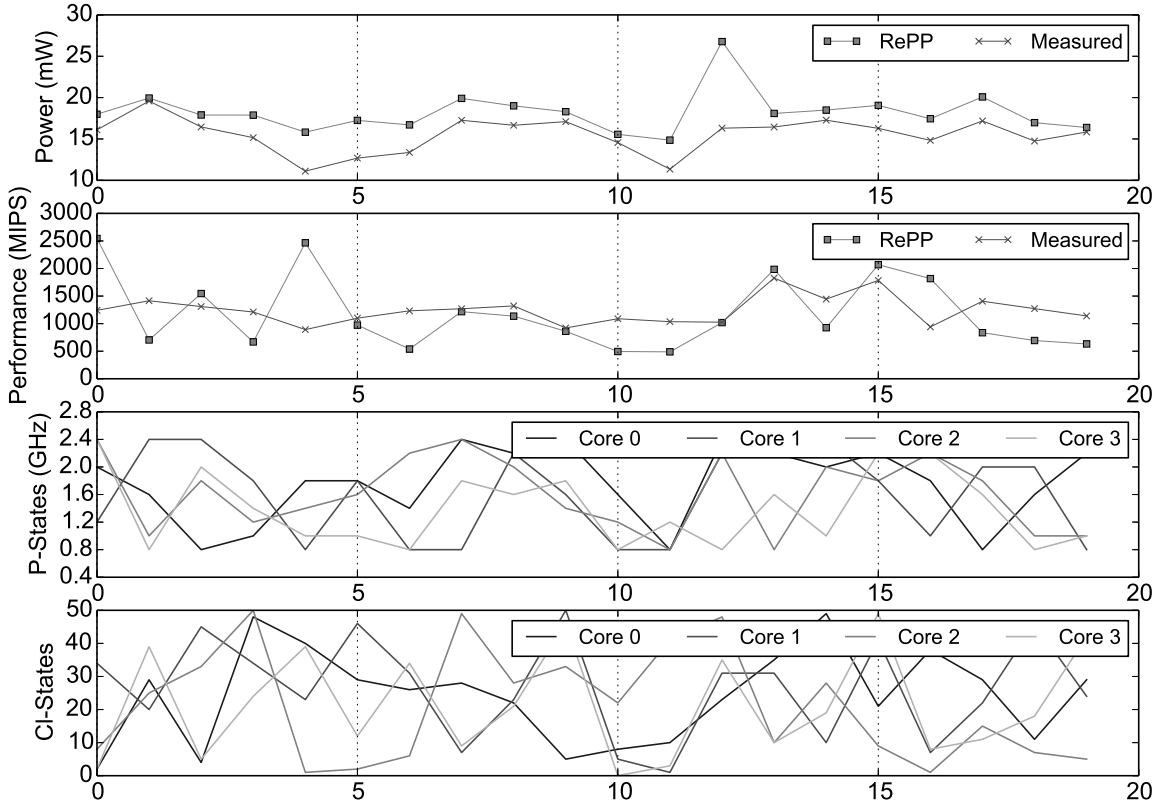


Figure 4.10 REPP-H for workload SSSN. Runtime power and performance prediction over time (in seconds) for workload SSSN.

Figure 4.10 shows an example of the power and performance prediction in runtime implemented on our system for the first 20 seconds of execution for the workload, SSSN. Specifically the workload SSSN consists of benchmarks `milc`, `milc`, `xalancbmk` and `blackscholes`. From top-to-bottom, the first (and second) graph represents the power (and performance) as measured using RAPL (and PMC) and the prediction made using REPP-H. The third and fourth graphs show the random combination of P-States and Cl-States generated for individual cores, respectively, for the first 20 seconds. We highlight two results. First, REPP-H does show the capability to adapt to workloads consisting of multiple thread phases (SPECcpu2006 and PARSEC 3.0 benchmarks have both memory and computational bound phases). For instance, observe at second 12, REPP-H makes a 11 mW error in predicting power, this is because of the huge changes in P-States and Cl-States. In this scenario, the P-States for core zero, one, two, three change from 0.8 to 2.4, 0.8 to 2.2, 0.8 to 2.2 and 1.2 to 0.8 respectively and the Cl-States change from 10 to 23, 1 to 31, 41 to 48 and 3 to 35. Observe that these errors only occur with huge changes in P-States and Cl-States in rapid intervals (for example, second four). Ozlem et al [87] on the other hand, show that rapid changes in power or performance are seldom required in data centre environments. Second, REPP-H can predict power and performance per thread, which can not be accomplished using the in-built RAPL register. In this particular workload, we make an error of 9.4% (384 mJ) and 15.2% (1500 MIPS) when predicting power and performance over 300 seconds, respectively.

Figure 4.11 shows the energy consumption in millijoules (mJ) using our prediction technique, REPP and the power measured using native RAPL register for all workloads over a period of 300 seconds when switching across 1000 combinations of P-States and Cl-States. On average, workloads incur an error in prediction of 8.6% or 332.31 mJ. Observe that the maximum error we incur is 12.1% (507.36 mJ) in workload SSFN. Moreover, the error in predicting power without considering insensitive, cache-friendly, cache-fitting and

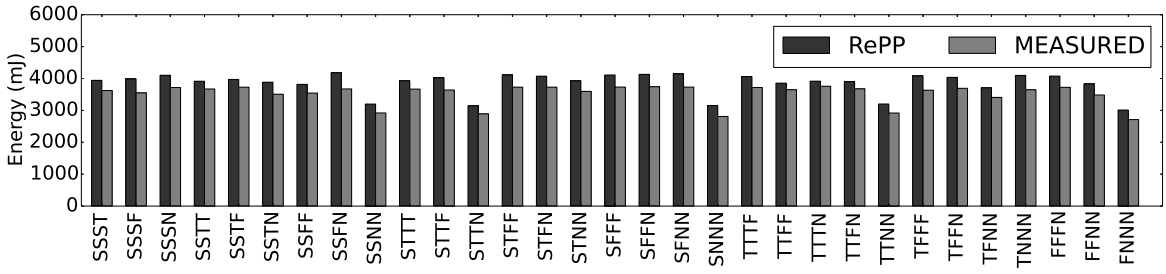


Figure 4.11 Power prediction for multiprogrammed workloads. *Energy consumed (*mJ*)* across all workloads on Intel processor. The *y*-axis is read as predicted (REPP-H) and measured (using RAPL).

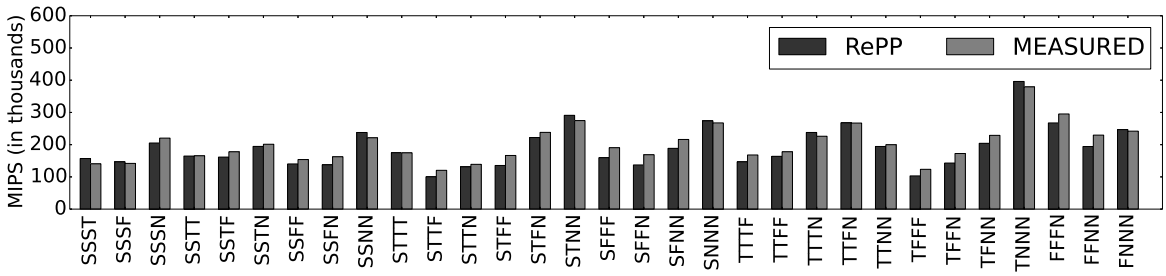


Figure 4.12 Performance prediction for multiprogrammed workloads. *Total performance (in thousands)* across all workloads on Intel processor. The *y*-axis is read as predicted (REPP-H) and measured (using PMCs).

thrashing benchmarks is 8.2% (326.60 mJ), 8.7% (319.79 mJ), 9.6% (367.42 mJ) and 8.2% (306.92 mJ), respectively

Figure 4.12 shows the total performance (MIPS) in thousands using our predicting technique (REPP-H), and the performance `measured` using PMCs for all workloads over a period of 300 seconds when switching across 1000 combinations of P-States and Cl-States. The average error we incur is 8.8% (over the 1000 combinations on all cores) and the maximum error is 18.8% in workload SFFN. Note, while disregarding insensitive benchmarks, the rest of the workloads have an error of 10.2%, and on the other hand not considering thrashing benchmarks the remainder of them have an error of 7.5%. Similarly, not considering cache friendly or cache fitting workloads produces an error of 9.8% or 9.9%, respectively.

Figure 4.11 and 4.12 demonstrate that REPP-H has been validated under multiple combinations of P-States and Cl-States across 35 multiprogrammed workloads. Moreover, each bar in the histogram represents a summary of the workload execution, similar to Figure 4.10.

4.2.2.4 Evaluating multi-core models including contention with constraints

In this section, we validate REPP-H by predicting performance, and power at all configurations (Section 4.1.1.4), and then select a configuration based on the user provided constraint. This constraint is defined as either delivering a minimum performance, or not violating a power target. Therefore to satisfy either of these constraints, REPP-H provides a dynamic configuration selector.

REPP-H configuration selector — Modern data centres have power constraints (e.g., power capping), and run multiple application instances with different performance constraints. This requires an algorithm to select a configuration per core such that the performance and power constraints are met per core and/or per application. To ensure that these constraints

are met, REPP-H dynamically selects a configuration per core every 250 ms by performing a linear search for the P-State that is the nearest to the given constraint; next, REPP-H selects the Cl-State for the given P-State that satisfies the constraint. This selected configuration, P-State, Cl-State, is used to ensure that the constraint is met for each interval. This process is repeated across all cores at the same time.

In our study, the performance and power constraints are given at a system level. These constraints are distributed homogeneously across all cores. For example, if an AMD server can only consume 600 W, that power constraint is distributed 25% per core, allowing each core to consume 150 W (it would be 50% on ARM). At runtime, for the spawned applications, REPP-H samples application behaviour periodically and uses the models built to predict performance and power at all configurations. REPP-H then selects the configuration for each interval to satisfy the local constraint.

This ability to satisfy constraints per core makes REPP-H better than previous works [71, 68, 85, 86, 58], allowing for multi-node, multi-core data centres running numerous application to satisfy a wide-range of performance and power requirements.

Experiments — We perform two types of experiments: one for validating the power capping mechanism and the other for delivering a minimum performance. We define two input parameters: (a) frequency of change, and (b) χ, which represents load or power. The average load offered by the applications is constant between two load changes, which can occur every **load_change interval** (1, 6, or 9 seconds), based on a **change_factor**, as follows. Load starts at a minimum, and varies by multiplying load by change_factor τ until it reaches a maximum load χ_{max}; thereafter, the load is multiplied by the negative value of change_factor until it reaches the minimum χ_{min}.

The values of change_factor tested were 20% (**Low**), 35% (**Mid**), and 50% (**High**). The minimum load is defined as the sum of smallest IPS for all four applications running at minimum frequency; similarly, maximum load is the sum of highest IPS for all four applications at maximum frequency. In another set of experiments, we change the power consumed by the workload similar to the load offered by the workload.

Mathematically, the experiment conducted for a load_change interval 1 can be represented as follows: In Equation 4.7, ψ represents the number of datapoints before the maximum load/power (χ_{max}), and in Equation 4.8, $\chi(\kappa)$ shows the datapoint κ in the sequence.

$$\psi = \left\lfloor \frac{\log(\chi_{max}/\chi_{min})}{\log(1+\tau)} \right\rfloor \tag{4.7}$$

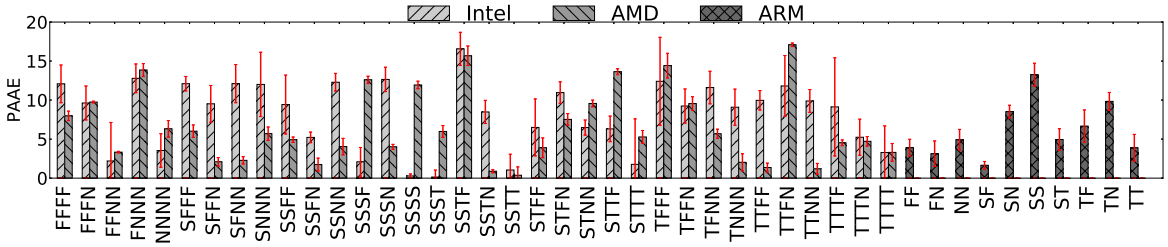
$$\chi(\kappa) = \begin{cases} \chi_{min} \times (1+\tau)^{n} & 0 \leq n \leq \psi \\ \chi_{max} \times (1+\tau)^{2\psi - n} & \psi < n \leq 2\psi \end{cases} \tag{4.8}$$

The error occurs when REPP-H selects a configuration that makes the application fall short of the minimum required performance (or exceed the maximum power requirement) in a given mapping interval.

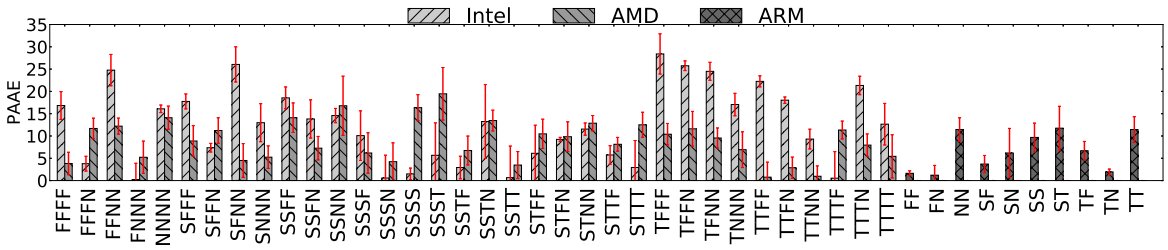
We ran ten experiments for power and ten for performance. Nine of them come from the combinations of the two parameters (frequency of change and load/power) described above; the tenth comes from a **Random** setting within fixed boundaries of either power or performance. Selecting a broad spectrum of load (or power) and frequency of change allowed us to validate REPP-H across multiple combination of configurations at runtime.

Evaluation — We present the results, and evaluate REPP-H when predicting performance and power on multicore processors by selecting a configuration in a single step to meet power and performance requirements for 35 multiprogrammed workloads across 20 experiments (ten for power and ten for performance). The prediction error is computed over a period of 400 seconds for all P-States, and Cl-States. The error metric indicates that the constraint was violated by that amount, which occurs when REPP-H selects a configuration that makes the application fall short of the minimum required performance (or exceed the maximum power requirement) in a given mapping interval.

Figure 4.13 shows the average PAAE for each workload when meeting the performance (ARM 7.1%, AMD 9.02%, and Intel 7.1%) and power (ARM 6.0%, AMD 6.6%, and Intel 8.1%) requirements in ten experiments across architectures. The error bars represent the STDEV across ten experiments for power and performance, which is less than 5.3% for each workload. We focus on analysing those workloads with an error greater than 10% in both architectures. When predicting power, we observe an error of 13.8% (654.86 mJ) on AMD for workload `FNNN` that contains `ep.C`, which has very high activity ratio in BPU (4.2542) and FE (13.752). Similarly for workload `TTFN`, Intel has a higher error 11.8% (11.34uJ) than AMD, because on Intel we run two instances of `radix`, whereas on AMD we run a single one. Recall that the multiprogrammed are generated based on the methodology described in Section 3.4.1, therefore we can not control the number of instances in a given workload. When predicting performance, we observe an error of 26.0% for `SFNN` on Intel because we run `lu_ncb`, which has non contiguous blocks of memory and the activity ratio in LLC is higher relative to other benchmarks. Similarly, workloads `TTTF` on AMD and `TTFN` on Intel have a power prediction error of 11.3% and 18.0%, respectively, because `radix` is part of the multiprogrammed workloads. The maximum performance error on AMD, ARM and Intel

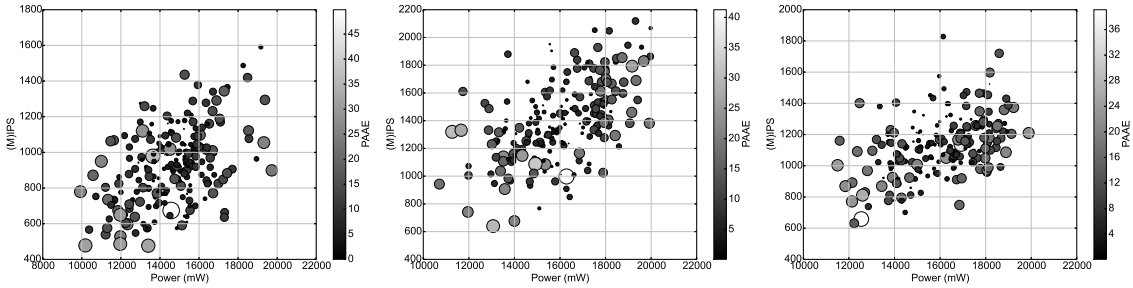


(a) Power



(b) Performance

Figure 4.13 Average PAAE for REPP-H with multiple constraints. Average PAAE when predicting power (4.13a) and performance (4.13b) for all workloads under Low, Mid, High and Random change_factors in multicore architectures. The error bars represent STDEV across change_factors. The *x*-axis shows multiprogrammed workloads.



(a) SSTN **(b)** FFFN **(c)** STFN

Figure 4.14 PAAE for workloads SSTN, FFFN, STFN on Intel. Performance and power prediction error with change_factor High.

are 19.4% (769 MIPS for SSST), 11.7% (5389 MIPS for ST) and 28.4% (13611 MIPS for TFFF). Similarly the maximum power error AMD, ARM and Intel are 17.0% (101.72 mJ for TTFN), 13.3% (10 mJ for SS) and 16.6% (37.24 mJ for SSTF), respectively.

Figure 4.14 represents the performance on *y*-axis and power on *x*-axis for multiprogrammed workloads SSTN, FFFN, and STFN on Intel architecture with *change_factor* High. The radius of each circle is the maximum prediction error, either power or performance (that is, radius = max($PAAE_{power}$, $PAAE_{perf}$)). There exists multiple grayscale grading from low PAAE (*black*) to high PAAE (*white*). Although the maximum PAAE shown is at the 50%

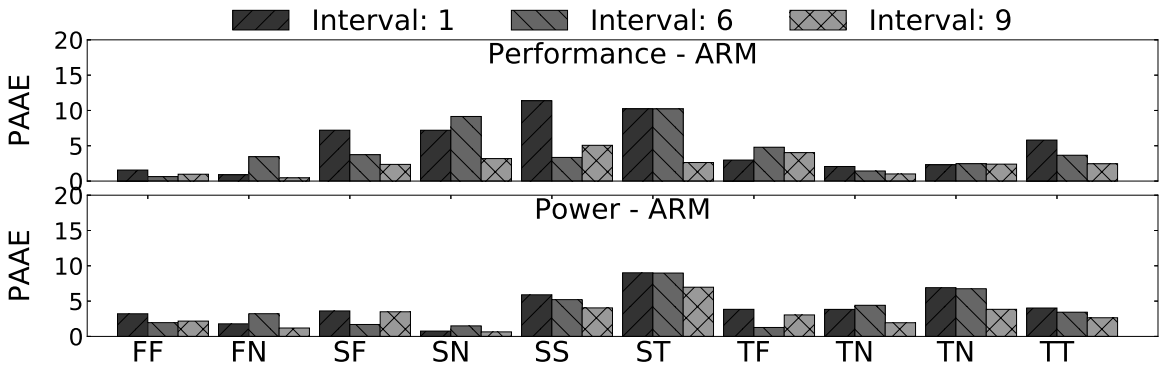


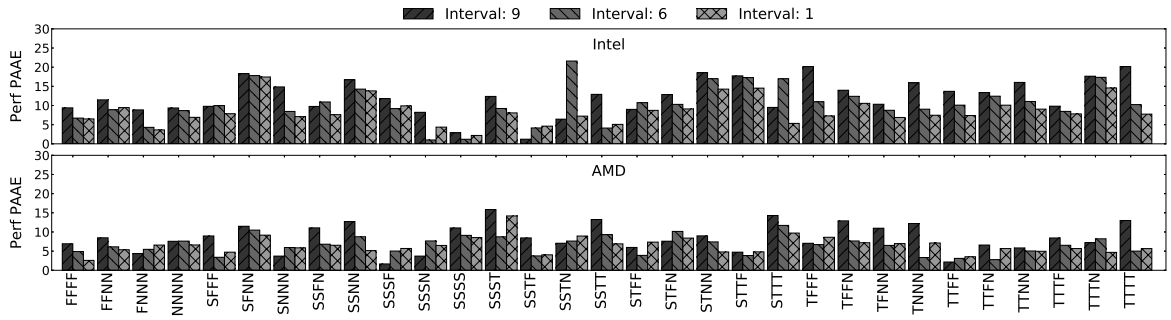
Figure 4.15 Average PAAE on ARM under different load_change intervals. Average PAAE when predicting power and performance for all workloads under different load_change `intervals` for change_factors Low, Mid and High. The *x*-axis shows multiprogrammed workloads.

mark (the grayscale in Figure 4.14a is up to 50%), the number of error predictions with PAAE greater than 30% is less than ten. For SSTN, FFFN and STFN, the average error when predicting power (and performance) of 9.7% (3.3%), 8.7% (9.4%) and 8.4 (6.7%). This behaviour is observed across all workloads.

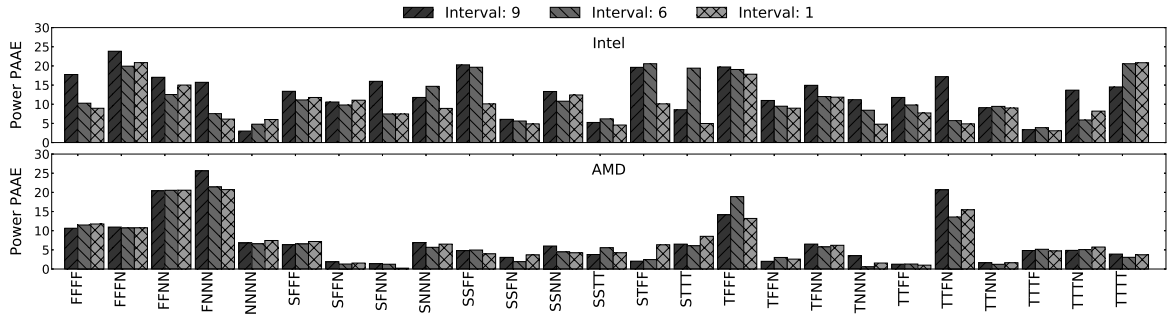
Figures 4.16 and 4.15 show the average PAAE for each workload under different load_change `intervals` on ARM and Intel/AMD when predicting performance (Figure 4.16a) and power (Figure 4.16b). Figures 4.15 and 4.16 are separated because ARM has different number of cores. Across the three architectures, faster (interval=1) load_changes have 3.5% higher error compared to slower (interval=9) load_changes because of aggressive changes in load in short burst cause numerous changes in configuration, thus leading to a higher error. On the other hand, slower load_changes lead to fewer changes in configuration, and have stable phases in application behaviour. Fortunately, such rapid changes in load seldom occur in data centres [17, 71]. Table 5.1 summarises the result obtained when predicting power and performance for load_change interval 1, 6 and 9.

Table 4.10 Average PAAE for load_change intervals. Power and performance error for load_change intervals 1, 6 and 9 on Intel, AMD, and ARM.

	Power			*Performance*		
Interval	1*s*	6*s*	9*s*	1*s*	6*s*	9*s*
Intel	11.1%	7.3%	5.6%	11.4%	10.6%	8.7%
AMD	7.2%	6.9%	6.7%	8.5%	6.5%	6.5%
ARM	4.2%	3.8%	2.9%	5.1%	4.6%	2.4%



(a) Performance



(b) Power

Figure 4.16 Average PAAE on Intel, and AMD under different load_change intervals. Average PAAE when predicting power and performance for all workloads under different load_change `intervals` for change_factors Low, Mid and High. The *x*-axis shows multiprogrammed workloads.

Enabling Power/Performance Capping — Determining a configuration to meet the minimum performance (or not exceeding power consumption threshold) is usually done in an iterative fashion (as in the RAPL driver on Intel processors [13]). A feedback control loop is often used to determine the configuration. If the power usage is above a certain threshold, the configuration is lowered. On the other hand, if the power usage is below a certain threshold, the configuration is increased to improve performance. In contrast, we provide a single-step mechanism to select configurations.

Figure 4.17 shows the responsiveness to (dynamic) power capping for workload `SSTT`, where the power capping limit is Random on Intel platform in the first 40 seconds. `SSTT` is composed of applications `streamcluster`, `lu.C`, `bwaves` and `soplex`. REPP-H changes P-States and meets the power target in 0.37 seconds on average, which is 3.6× faster than the iterative algorithm (used by Intel RAPL). This time includes sampling interval, latency to predict at all P-State and Cl-State and time to change P-State. Moreover REPP-H, provides 6% higher prediction accuracy.

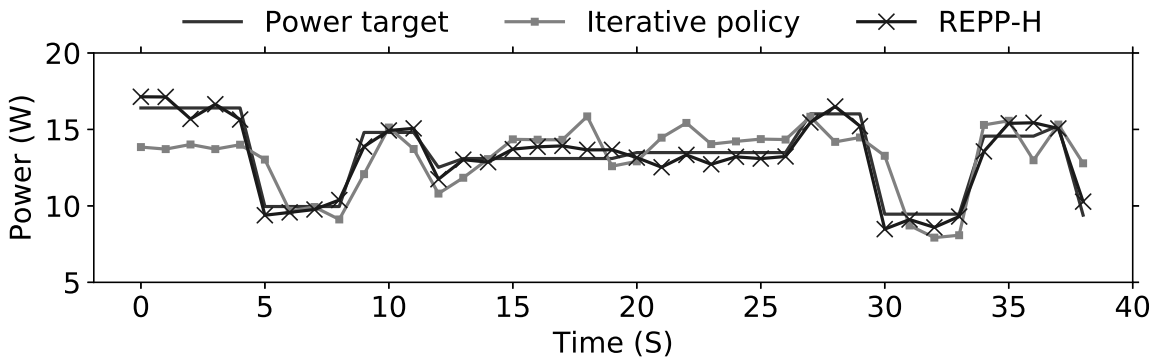


Figure 4.17 Responsiveness to power change on Intel. Responsiveness to power capping for workload SSTT with constraint Random.

4.2.3 Conclusion

In this section, we introduced REPP-H, a scalable power, and performance modelling, and prediction technique to meet various user-defined criteria. REPP-H is built on existing real-hardware, and uses the hardware support currently available to profile the behaviour of applications, which is not known prior to execution, on modern multicore systems.

We validate REPP-H using several single threaded and multiprogrammed workloads with average errors, respectively, on ARM, AMD and Intel architectures of 7.1%, 9.0%, 7.1% when predicting performance, and 6.0%, 6.5%, 8.1% when predicting power consumption. Moreover, the single-step prediction technique provided by REPP-H is 3.6× faster than iterative algorithms. We argue that REPP-H can enable operators to better control power and performance in modern data centres that include server architectures with heterogeneous processing capabilities.

4.3 REPP-C: Runtime Estimation for Performance and Power with Workload Consolidation

In the previous section, we introduced REPP-H, a power and performance modelling and prediction technique for multicore architectures. Our results have shown REPP-H has a high accuracy in predicting performance, and power. In this section, we extend the model built to predict in multicore environment with workload consolidation. Our approach in implementing REPP-C leverages motivation from prior research which show that a combination of prediction, and scheduling techniques, that avoid negative interference, can minimise power consumption.

nishtala: citations are missing

4.3.1 FACTS

need pseudocode

4.3.2 REPP-C

4.3.3 Evaluation

nishtala: **Implementation**: power when shutoff, and idle

4.3.3.1 FACTS

4.3.3.2 REPP-C

nishtala: Need psuedocode to fix this !

4.4 Implementation

- MSR registers to change frequency.
- cost to use.
- can speak about how much it costs to change using just the module
- talk about scalability

nishtala: need a summary table for REPP

PART III:

ADDRESSING SCHEDULING OF INTERACTIVE, AND BATCH WORKLOADS

> Errors using foo barbazinadequate data are much less than those using no data at all.
>
> CHARLES BABBAGE

CHAPTER 5

Hipster

nishtala: subscript always with mathit, and not textrm

Prior chapter 4 introduced an approach of performance, and power modelling to find the bad text sweet spot between performance and power. By contrast this chapter focuses on a thread-level scheduling approach that couples the mapping applications such as latency-critical, and batch workloads on *heterogeneous* ARM processor while selecting a DVFS setting to optimise energy-efficiency. Coupling latency-critical workloads with batch workloads is interesting because, the former is striving towards delivering a service within a certain quantum, else leading to a violation, whereas the latter is aimed at maximising performance.

There has been significant amount of research [88, 33, 31, 31, 30, 89, 5, 90–92] carried out in the area of mapping thread-to-cores based on application level metrics. Nevertheless, these scheduling algorithms are optimised based on only the previous time quantum, thereby making a reactive approach, and do not learn from prior violations for latency-critical workloads or select a mapping that improves energy-efficiency.

By contrast to prior approaches, we introduce **Hipster**, a hybrid reinforcement learning (RL) approach coupled with a feedback controller that dynamically allocates workloads to heterogeneous cores while selecting optimised DVFS settings. We propose a variant, called HipsterIn, that is optimised for latency-critical workloads running solo in the system, adjusting the system configuration to reduce energy consumption. The HipsterCo variant, which supports collocation of latency-critical and batch workloads, and focuses on maximising the throughput of the batch workloads. Both variants of Hipster always ensure that the QoS requirements are met for the latency-critical workload.

nishtala: need to organize everything below

nishtala: lambda replaced with phi

nishtala: gamma remains gamma

nishtala: alpha replaced with xi

nishtala: where and when should octoman be explained?

5.1 Methodology

In this section we describe the hybrid reinforcement learning approach that combines reinforcement learning with heuristics by first introducing Hipster reinforcement learning approach. Next we introduce design of hipster, the heuristic mapper, and the reward calculation method for the reinforcement learning mechanism. Finally, we introduce the mechanism used to map threads-to-cores based on the reward mechanism.

5.1.1 Hipster Reinforcement Learning

The RL problem solved by Hipster is formulated as a Markov Decision Process (MDP). In an MDP, a decision-making process must learn the best course of action to maximise its total reward over time. At each discrete instant, n, the process can observe its current *"state"*, w_n, and it must choose an *"action"* c_n from a finite set of alternatives. Depending on the chosen action and current state (but nothing else), there is an unknown probability distribution controlling which state, w_{n+1}, it enters next and the reward, ϕ_n, that it receives. The problem is to maximize the total discounted reward, given by $\sum_{n=0}^{\infty} \gamma^n \phi_n$, where γ is the discounting factor. The discounting factor γ should be positive and (slightly) less than one, in order to reflect a moderate preference for rewards in the near future.

The hybrid task management problem solved by Hipster is translated to an MDP as follows. The state w_n indicates the current load[1] on the latency-critical workload, measured during the (prior) time interval t_{n-1} to t_n. Hipster quantises the load into buckets. Specifically the latency-critical application provides a measurement of the percentage load during the time interval, in terms of requests per second, queries per second, or similar. The action, c_n, which is chosen by Hipster depending on the state, determines the configuration to be used in the (next) time interval, t_n to t_{n+1}; that is, the combination of cores and DVFS settings allocated to the latency-critical application. These settings are used for the upcoming interval, at the end of which, at time t_{n+1}, the reward ϕ_n is determined depending on the level of QoS relative to the target, given a metric of optimisation: either the system power consumption (HipsterIn) or the throughput of the batch workloads (HipsterCo). A precise definition of the calculation of the reward is given in Section 5.1.4.

[1] Load is Chapter 5 refers to load of the latency-critical workload measured in requests per second, or queries per second, In Chapter 4 it is referred to as instructions per second.

RL is a type of unsupervised machine learning with a focus on online learning [93]. It solves an MDP by maintaining a table of values, $R(w,c)$, indexed on the possible states $w \in W$ and possible actions $c \in C$. The entry $R(w,c)$ estimates the total discounted reward that will be received, starting from state w, if the decision-making process starts by choosing next action c. Assuming that the **lookup table**, $R(w,c)$ has close to correct values, then, if the current state is w_n, the best action c_n is the one that gives the largest total discounted reward; i.e. $c_n = \arg\max_c R(w_n,c)$. The process chooses this value of c_n, then it updates $R(w_n,c_n)$ using a particular formula based on the old and new states, w_n and w_{n+1}, and the reward ϕ_n.[2] A classic problem in RL is known as the *exploitation–exploration dilemma*, which captures the need not only to exploit the best solution identified so far, but also to fully explore alternatives, which may or may not be better.

Hipster uses a hybrid RL approach [94], which combines reinforcement learning with a heuristic, to be used while the algorithm is still learning the optimal behaviour. For Hipster, the heuristic improves QoS at the beginning of the execution and it is also re-used after a change in the characteristics of the problem, e.g. the mix of batch workloads. A hybrid RL [94] has the potential to outperform pure RL schemes [95, 96] that only deal with the exploitation–exploration dilemma (e.g. Q-learning), for several reasons:

(1) During the learning phase, online unsupervised learning without a heuristic generates random decisions, which would produce an unacceptable number of QoS violations.

(2) As the complexity of the problem increases, in terms of workloads, number of cores, DVFS settings, and so on, it may take longer to learn the table R. In contrast, a hybrid RL can find acceptable solutions even during the learning phase.

(3) The exploration feature of many RL approaches is necessary to capture a global maximum, but it may cause extra QoS violations. Using a heuristic in the learning phase can reduce the need to explore configurations that clearly violate QoS.

5.1.2 Hipster Design

Figure 5.1 shows a high-level view of Hipster. Hipster includes a QoS Monitor, a Learning Phase and an Exploitation Phase. Given a QoS target, an incoming load, and a metric to optimise for, Hipster learns the most adequate core configuration and DVFS settings by managing a lookup table that is used to map the workloads to the available hardware resources.

[2] The update of $R(w_n,c_n)$ is on line 16 of Algorithm 1.

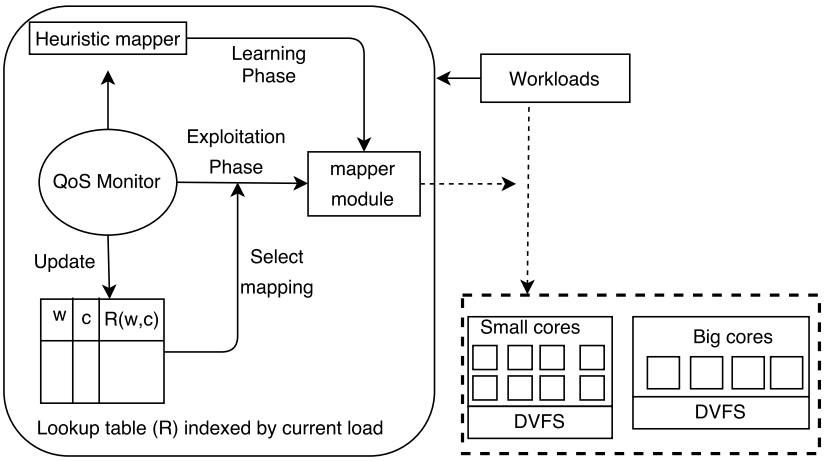


Figure 5.1 High-level view of Hipster runtime system

QoS Monitor — The QoS Monitor is responsible for periodically collecting the performance statistics from the latency-critical and batch workloads. For the latency-critical workload, Hipster gathers the appropriate application-level QoS metrics such as throughput (RPS or QPS) and latency (query tail latency). It also reads the current load on the latency-critical workload and quantises this value into discrete buckets between 0 and $T-1$, for (some) small value T. HipsterCo uses a profiling tool to measure the throughput of the batch workloads, using per-core hardware performance counters, such as CPU utilization, cache-misses and IPS.

Learning and exploitation phases — The data collected by the QoS monitor is used to make the thread-to-core mapping decisions. In the learning phase, Hipster uses a feedback control loop based on heuristics to map the latency-critical workload to resources. Following the intuition from Section **??**, when load is low, the mapper executes the latency-critical workload on small cores at lower DVFS states, and when load is high, it uses a combination of big and small cores at higher DVFS. Hipster also begins populating the lookup table so that each entry approximates the corresponding total discounted reward. Specifically, Hipster uses the reward mechanism (Section 5.1.4) to prefer core configurations that minimise system energy consumption or maximise batch workload throughput, while ensuring as well as possible that at least 95% QoS guarantee is achieved [97].

In the exploitation phase, Hipster uses the lookup table to select the core mapping and DVFS settings, based on the load. It also continues to update the values in the lookup table, in order to continue to improve the mapping decisions. At runtime, Hipster determines when to dynamically switch between the learning and exploitation phases, based on a prefixed time

quantum. At deployment stage, we ensure that the bucket size for each workload gives at least 95% QoS guarantee [98] with minimal energy consumption.

5.1.3 Heuristic Mapper (Learning Phase)

The heuristic mapper is a state machine with a feedback control loop. The current state identifies the core configuration: the DVFS settings and number and type of cores to use for the latency-critical workload.[3] The choice of available states depends on the platform; that is, the total number and types of cores, and the DVFS settings. There is a predefined ordering of the states, approximately from highest to lowest power efficiency. This ordering is determined by measuring the power and performance of each state using a stress microbenchmark consisting of mathematical operations without memory accesses.

Whenever QoS is close to being violated, the state machine transitions into the next-higher power state. The QoS is quantified using the currently measured tail latency at the 95th or 99th percentile, denoted QoS_{curr}. The target tail latency is denoted by QoS_{target}. The state machine transitions to the next-higher state whenever the time interval ends in the so-called *danger zone* defined by:

$$QoS_{\text{curr}} > QoS_{\text{target}} \times QoS_{\text{D}}$$

where QoS_{D} is a parameter between 0 and 1 that defines the size of the danger zone. Whether such a state transition improves or degrades performance and whether it actually increases or decreases power depends on the characteristics of the platform and the particular workloads. The state machine may have to make several consecutive state transitions until the QoS is met.

In contrast, whenever the QoS is far from being violated, the state machine transitions into the next-lower power state. This happens whenever the time interval ends in the so-called *safe zone* defined by:

$$QoS_{\text{curr}} < QoS_{\text{target}} \times QoS_{\text{S}}$$

where QoS_{S} is a parameter between 0 and QoS_{D} that defines the size of the safe zone. The values of QoS_{D} and QoS_{S} are determined to avoid oscillations between adjacent states. In particular, QoS_{D} is empirically computed in the same way as for Octopus-Man [33, 97].

nishtala: how should it be written.. where is octoman shown?

[3] State machines and Markov Decision Processes use "state" with different meanings. In Section 5.1.3 (only), "state" refers to the core configuration, elsewhere it is the load.

Algorithm 1 Reward mechanism

▷ *Determine reward ϕ_n based on interval $t_n \ldots t_{n+1}$*

1 Let QoS_{target} be the target QoS of the interactive workload.
2 $QoS_{\text{curr}} = QoSMonitorLatency$
3 $Power = QoSMonitorPower$
4 $QoS_{\text{reward}} = QoS_{\text{curr}}/QoS_{\text{target}}$
5 $Power_{\text{reward}} = TDP/Power$ ▷ *TDP (thermal design power)*
6 **if** $QoS_{\text{curr}} < QoS_{\text{target}} \times QoS_{\text{D}}$ **then**
7 $\quad \phi_n = QoS_{\text{reward}} + 1$
8 **else if** $QoS_{\text{curr}} < QoS_{\text{target}}$ **then**
9 $\quad \phi_n = QoS_{\text{reward}} + 1 - Random(0,1)$
10 **else**
11 $\quad \phi_n = -QoS_{\text{reward}} - 1$
12 **if** there exist batch jobs **then**
13 $\quad \phi_n = \phi_n + \frac{B_{\text{IPS}}+S_{\text{IPS}}}{maxIPS(B)+maxIPS(S)}$
14 **else**
15 $\quad \phi_n = \phi_n + Power_{\text{reward}}$
16 $R(w_n,c_n) = R(w_n,c_n) + \xi\left(\phi_n + \gamma \max_{d\in C} R(w_{n+1},d) - R(w_n,c_n)\right)$

The heuristic proposed by Octopus-Man is attractive because of its simplicity but it can be sub-optimal (see Section **??**, Figure **??**) because there is no common static ordering of configuration states that works for all workloads. Moreover, in practice, the state machine may respond slowly to rapid changes in load. Nevertheless, we found that such a state machine heuristic is suitable to accelerate the learning phase of the RL algorithm by exploring viable core configurations to quickly populate reasonable values into the lookup table.

5.1.4 Reward Calculation

During both the learning and exploitation phases, the values in the lookup table are dependent on the reward, which is calculated as defined in Algorithm 1. This reward calculation is invoked after each monitoring interval, and its definition was determined empirically (more details in Section 5.1.6). The reward ϕ_n has three parts: the QoS Reward, Stochastic Reward, and either the Power Reward (for HipsterIn) or the Throughput Reward (for HipsterCo):

QoS Reward — The ratio of the measured QoS to the QoS target is known as QoS_{reward}. If this value is less than one, then the QoS target has been met, and it quantifies how quick the response was as the **QoS earliness**. In this case, line 7 or 9 applies a positive reward that prefers configurations that approach the QoS target, which acts as a heuristic to reduce energy consumption or improve batch workload throughput. If QoS_{reward} is greater than one, then the QoS target has *not* been met, and it determines how intense the violation was as the **QoS tardiness**. In this case, line 10 applies a negative QoS reward.

Stochastic Reward — When the QoS is below the target, as defined in Section 5.1.3, but still over the danger zone, then a stochastic penalty is applied (line 9 of Algorithm 1). The stochastic penalty offers the possibility to continue to explore the configuration, but with a smaller probability. In future, other external influences for the latency-critical workload like noise, contention on shared resources, pending queue lengths, etc., may cause a QoS violation.

Power Reward (HipsterIn) — The ratio of the thermal design power (TDP) to the measured system power consumption is known as $Power_{\text{reward}}$ as shown in line 15. A smaller value of this term means that the system power consumption was lower, and it increases the reward.

Throughput Reward (HipsterCo) — Lines 12 to 13 of Algorithm 1 calculate the Throughput Reward, which is approximately proportional to the total throughput of the batch workloads. Since HipsterCo does not require modifications to the batch workloads, it is only possible to measure their throughput in a generic way using performance counters. Specifically, the throughput is quantified in terms of IPS. The parameters B_{IPS} and S_{IPS} measure the total IPS of the big and small clusters running batch workloads, respectively. The denominator is constant given by the sum of $maxIPS(B)$ and $maxIPS(S)$, which measure the maximum IPS, at highest DVFS, for the big and small cores respectively. More details are given in Section **??**.

Once the reward ϕ_n has been calculated, line 16 updates the value of $R(w_n, c_n)$ in the lookup table, and this is done in the same way during both the learning and exploitation phases. This update is controlled using two scalar parameters, both between zero and one: the discounting factor, γ, and the learning rate, ξ.

Discounting Factor, γ — The γ coefficient in line 6 of Algorithm 1 is the discounting factor, which quantifies the preference for short-term rewards [99]. Setting $\gamma = 0$ means that the algorithm only relies on immediate short-term rewards. To allow a balance between short-term and future rewards, we set $\gamma = 0.9$ (empirically determined). In other words, this methodology allows the optimization problem to also take into account future rewards.

Learning Factor, ξ — The ξ coefficient in line 16 of Algorithm 1 is the learning factor, which controls the rate at which the values in the lookup table $R(w,c)$ are updated. A large value of ξ close to one means that the algorithm learns quickly, favouring recent experience, but increasing the susceptibility to noise. In contrast, a small value of ξ means that the algorithm learns slowly. In our experiments we used $\xi = 0.6$

Algorithm 2 Exploitation Phase

```
1  Let X be threshold on QoS guarantee to re-enter learning phase
2  Let w_n be observed load for interval t_{n-1} ... t_n
3  Let c_n be configuration for interval t_n ... t_{n+1}
4  Let R(w,c) = 0 for all w, c
5  Let n = 0
6  repeat
                                  ▷ At time t_n, choose configuration for t_n to t_{n+1}
7      Let c_n = max_{d∈C} R(w_n, d)
8      if there exist batch jobs then
9          Allocate unused cores to batch jobs
10         if latency-critical jobs on a single core type then
11             Set highest DVFS for other core type
12     else
13         Set lowest DVFS for unused cores
14     Sleep until t_{n+1}                    ▷ Run for interval t_n to t_{n+1}
15     Let w_{n+1} be the quantised load from the latency-critical workload
16     Call Algorithm 1                       ▷ Algorithm 1 updates R(w_n, c_n)
17     n = n + 1
18     if QoSGuarantee ≤ X then Learning phase
19 until Terminated
```

5.1.5 Exploitation Phase

The exploitation phase of Hipster is defined by Algorithm 2. Line 7 determines the configuration, c_n, with the highest estimated total discounted reward. Lines 8 to 13 apply the configuration by mapping the workloads to the resources, as described below, depending on the specific variant of Hipster (HipsterIn or HipsterCo). Line 14 runs the workload for the next time interval, and line 16 calls Algorithm 1 to update the lookup table, based on the metrics obtained by the QoS Monitor during the time interval. Line 18 re-enters the learning phase when necessary. The mapping of workloads to resources is as follows:

Reward Mechanism for HipsterIn — To minimise power consumption while meeting the QoS target for latency-critical workloads, the configuration with the highest reward is selected and then DVFS setting for the unused cores is set to the lowest value (Lines 12 to 13 of Algorithm 2).

Reward Mechanism for HipsterCo — Corroborating the findings of prior work [30], we observed that collocating both latency-critical and batch workloads degrades QoS at higher loads due to shared resource contention. If the reward mechanism were not aware of such collocations, it may make decisions that violate QoS for the latency-critical workload and/or reduce the throughput of the batch workloads. As a precursor to this condition, we introduce the following mechanisms. First, to maximise the throughput of the throughput-oriented

workloads while meeting QoS targets, all of the unused cores are allocated to the batch workloads (lines 8 to 9 in Algorithm 2). Second, in case the latency-critical job is allocated exclusively to a given core type, the other core type is set to the highest DVFS to accelerate the batch workloads (lines 10 to 13 in Algorithm 2). For instance, on a two-socket/cluster system with two cores per socket/cluster, if the latency-critical workload is running on two small cores, the big cores are allocated to the batch workloads at the highest DVFS.

5.1.6 Responsiveness and Stability

To ensure that QoS is met for latency-critical workloads, the scheduling policy must quickly respond to fluctuations in load and latency, either due to changes in core mapping, DVFS or any external influence. Therefore, the responsiveness and stability of Hipster is determined by (a) the computation latency in migrating cores and setting DVFS. (b) the reaction time of QoS between migrating an application from current mapping to future mapping, and (c) the granularity of monitoring for the latency-critical workload's QoS.

The computational latency for changes in core mapping and DVFS are negligible [100, 12, 101]. The default monitoring interval for Memcached and Web-Search is one second. Based on the aforementioned overheads, we determine the sampling interval as a sum of the monitoring interval for the latency-critical application, and the overhead to switch the core mapping and DVFS.

5.2 Evaluation

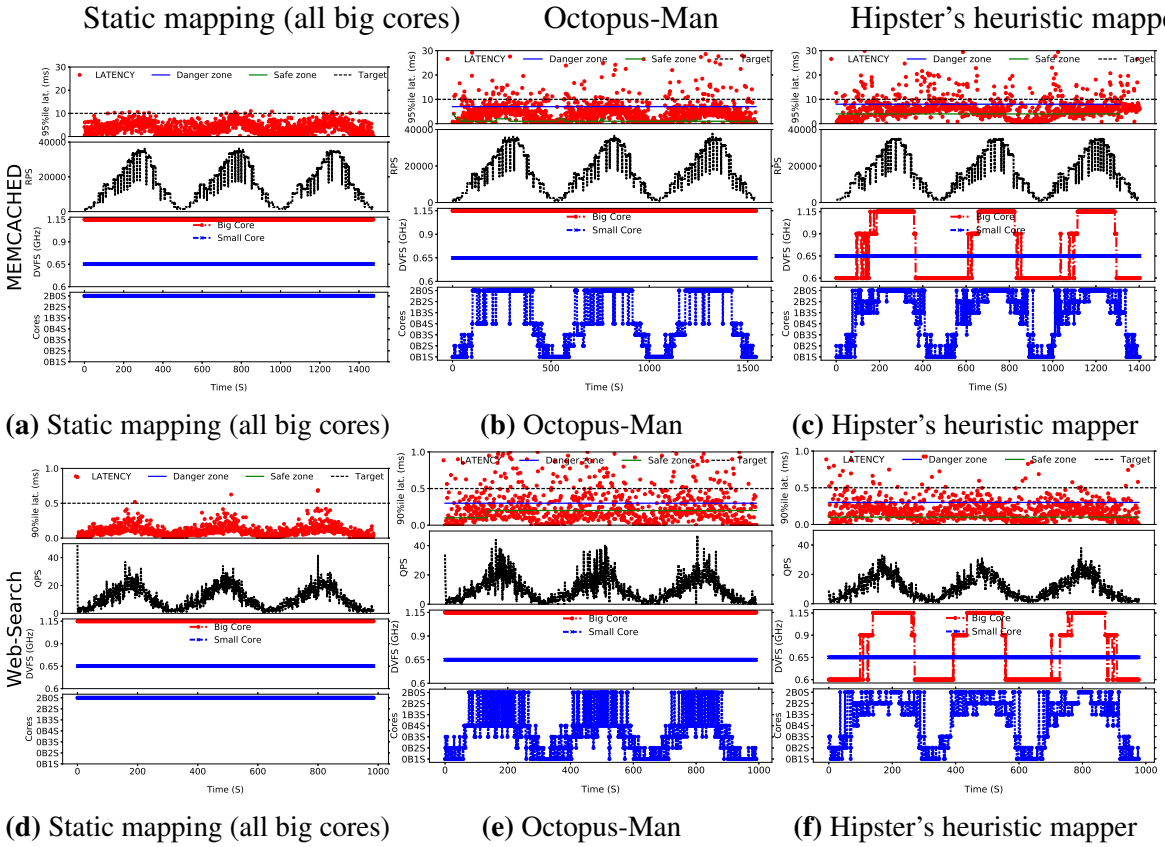
nishtala: need to review from here?

5.2.1 Algorithm configuration

In deploying Octopus-Man, we first performed a sweep on the danger and safe thresholds, and picked the combination of thresholds with the highest QoS guarantee. For HipsterIn, we set the learning phase to be 500 seconds, except when quantifying the learning time, where we set it to 200 seconds.

5.2.2 HipsterIn Results

This section evaluates the effectiveness of HipsterIn, as a policy for managing a single interactive workload. The objective is to minimise system energy consumption while satisfying QoS.



(a) Static mapping (all big cores) **(b)** Octopus-Man **(c)** Hipster's heuristic mapper

(d) Static mapping (all big cores) **(e)** Octopus-Man **(f)** Hipster's heuristic mapper

Figure 5.2 Comparison of heuristic policies. Hipster's heuristic policy (right-hand column) with static mapping (left) and Octopus-Man (centre). Results are shown for Memcached (top) and Web-Search (bottom) on ARM Juno R1.

5.2.2.1 Hipster's Heuristic Policy (interactive only)

We first evaluate the effectiveness of Hipster's heuristic policy alone, for mapping interactive workloads. Figure 5.2 shows the results for both workloads: Memcached (top row, subfigures (a), (b) and (c)) and Web-search (bottom row, subfigures (d), (e) and (f)). The columns, from left to right, correspond to static mapping, for which the interactive threads are mapped to the two big cores at highest DVFS of 1.15 GHz ((a) and (d)), Octopus-Man ((b) and (e)) and Hipster's heuristic policy ((c) and (f)). For each subfigure, from top to bottom, the first plot presents the tail latency (QoS), with the target marked with a dashed line. The second plot shows the achieved throughput in RPS (requests per second). The third plot presents the DVFS of the big and small cores, and the fourth plot represents the choice of core mapping.

Comparing the DVFS and core configuration subplots, we observe that Hipster's heuristic policy is successfully exploring the DVFS settings available on the Juno platform (third plots), and it is exploring all configurations including those that use both big and small cores

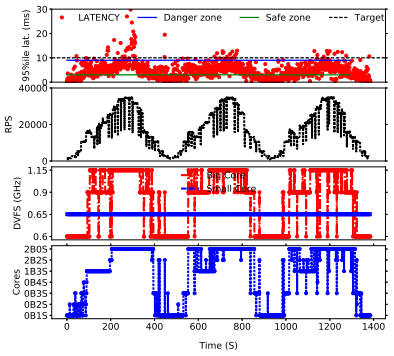


Figure 5.3 HipsterIn on Memcached

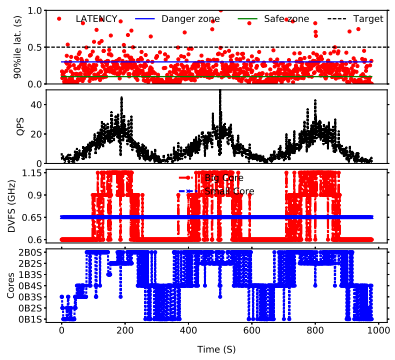


Figure 5.4 HipsterIn on Web-Search

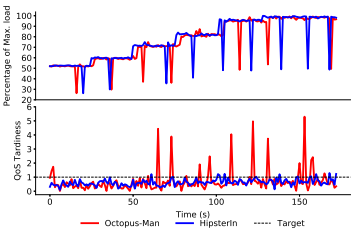


Figure 5.5 Percentage of Max. load, and Tail latency (QoS Tardiness) running Memcached with HipsterIn and Octopus-Man.

Table 5.1 HipsterIn: summary of QoS guarantees, tardiness and energy savings for Memcached and Web-Search.

	QoS Guarantee		*QoS Tardiness*		*Energy Reduction*	
	Memcached	*Web − Search*	*Memcached*	*Web − Search*	*Memcached*	*Web − Search*
Static (all big cores)	99.5%	99.5%	1.1	1.3	-	-
Static (all small cores)	85.8%	78.4%	1.4	2.0	48.0%	31.0%
Hipster's Heuristic	89.9%	95.3%	1.8	1.9	18.7%	13.6%
OctopusMan	92.0%	80.0%	2.2	2.1	17.2%	4.3%
HipsterIn	**99.4%**	**96.5%**	1.4	2.0	**14.3%**	**17.8%**

at the same time (bottom plots). In contrast, Octopus-Man does not adjust the DVFS settings and it uses either the big or small cores, but not both at once.

Both Octopus-Man and Hipster's heuristic policy frequently oscillate between consecutive core configurations. In the case of Octopus-Man, there are clear oscillations between two big cores and four small cores, for example between the 600^{th} and 800^{th} seconds. Using two big cores satisfies QoS but since it is within the safe zone, Octopus-Man switches to four small cores, which enters the danger zone, generating an alert provoking a return to two big cores. Such oscillations between cores in different clusters leads to severe QoS degradation of up to 20%. As expected, the static mapping (all big cores) has the least number of violations.

In summary, although Hipster's heuristic policy alone improves over Octopus-Man by exploiting a wider search space, it still suffers from an unacceptable number of QoS violations.

5.2.2.2 HipsterIn: Memcached Results

Figure 5.3 shows the results using HipsterIn for Memcached. After completing the learning phase, the oscillatory effect between core mappings is greatly reduced (by 8.3%), and overall

the QoS guarantee is improved by 24% compared with the learning phase. HipsterIn performs well because it moves directly to the appropriate core configuration for a given load that satisfies QoS.

In addition to switching between a combination of different cores, HipsterIn also explores more fine-grained DVFS adaptations, which has lower overheads (of microseconds) compared with migrations between cores (order of milliseconds) [31].

5.2.2.3 HipsterIn: Web-Search Results

Figure 5.4 shows the results using HipsterIn for Web-Search. In contrast to the heuristic policies (Octopus-Man and Hipster's heuristic), during the exploitation phase, HipsterIn monitors the QoS and dynamically adjusts the core mapping and DVFS settings to adapt to load fluctuations. Both Hipster's heuristic and Octopus-Man perform aggressive changes to core mappings to reduce energy, leading to a negative impact on QoS. On the other hand, HipsterIn shows a more balanced behaviour, performing 4.7× fewer task migrations than Octopus-Man for Web-Search, while improving QoS up to 16% and reducing energy consumption by 13.5%.

5.2.2.4 HipsterIn Summary

Table 5.1 summarises the QoS guarantee, QoS tardiness and energy reduction for Memcached and Web-Search for different policies: Static (all big cores), Static (all small cores), Hipster's heuristic mapper, Octopus-Man and HipsterIn. We compare the energy consumption of each mapping schema against Static (all big cores). We quantify the QoS behaviour at each sampling interval by assessing the measured QoS using two metrics: QoS guarantee and QoS tardiness.[4] The QoS Guarantee is the percentage of samples for which the measured QoS did not violate the target (100%-QoS violations%). The QoS tardiness in the table is the average (mean) of the QoS tardiness, including only the samples that violated the QoS target.

As shown in Table 5.1, for Web-Search and Memcached, static (all small cores) cannot meet the required QoS. On the one hand, the heuristic policies reduce energy marginally, but violate QoS due to excessive core migrations. On the other hand, HipsterIn meets QoS at 99.4% and 96.4% for Memcached and Web-Search, while having energy savings of 14.3% and 17.8%, respectively.

[4] QoS Tardiness is $QoS_{\text{curr}}/QoS_{\text{target}}$, using the definitions from Section 5.1.4.

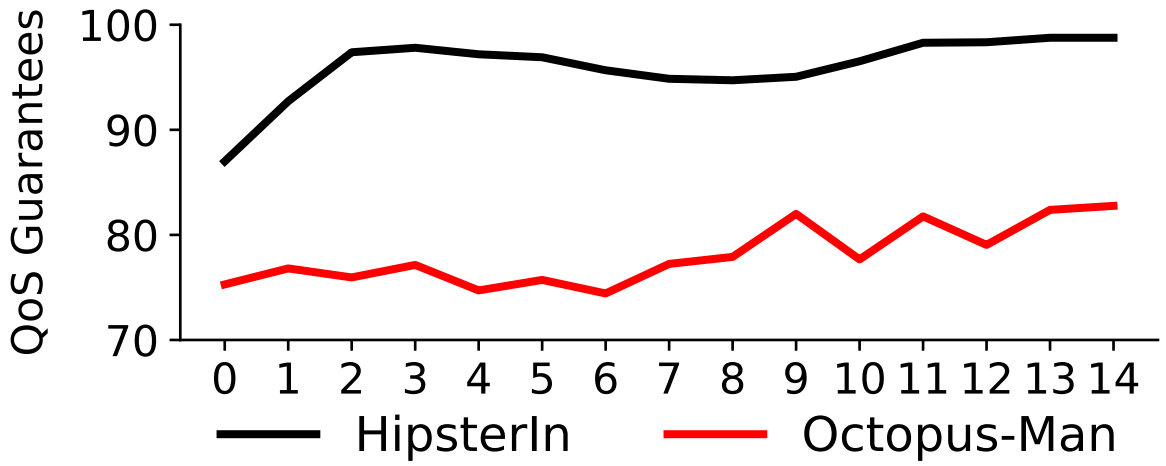


Figure 5.6 QoS Guarantees of HipsterIn and Octopus-Man. Each data point represents the QoS guarantees over 100 s intervals.

5.2.2.5 HipsterIn Analysis

Rapid adaptation to load changes — Hipster can respond to rapid changes in load by directly mapping to a configuration that satisfies the QoS. Figure 5.5 shows how HipsterIn (during the exploitation phase) and Octopus-Man respond to changes in load. From top to bottom, we express the input load in terms of the percentage of maximum load, where it increases from 50% to 100% over a period of 175 seconds for Memcached. In the second graph, we express the 95^{th} percentile tail latency as QoS Tardiness. A QoS violation has occurred if the QoS Tardiness is above 1, otherwise QoS is satisfied. We find that Octopus-Man violates QoS due to aggressive core mappings to minimize energy consumption. By contrast, HipsterIn achieves more stable tail latency even at higher load (80%). Note that, from 75% to 90% of the load, the QoS tardiness (extent of violation) experienced by HipsterIn is 3.7× (mean) lower than Octopus-Man.

Impact of learning time — HipsterIn aims to deliver the best balance between QoS guarantee and energy reduction compared to heuristic policies (Table 5.1). In practice, to best optimize for energy efficiency, and to improve QoS, HipsterIn needs a short learning phase. Figure 5.6 shows the QoS guarantee and energy distribution over 100 s intervals for Web-Search, for both HipsterIn and Octopus-Man. Each data point in the graph refers to a 100-second interval. The learning phase is set to 200 s. As can be seen, HipsterIn quickly learns during the heuristic phase, which improves QoS guarantees. On the other hand, for Octopus-Man, the QoS guarantees are consistently around the 80% mark, since it does not use past decisions and their associated effects to improve the future decisions.

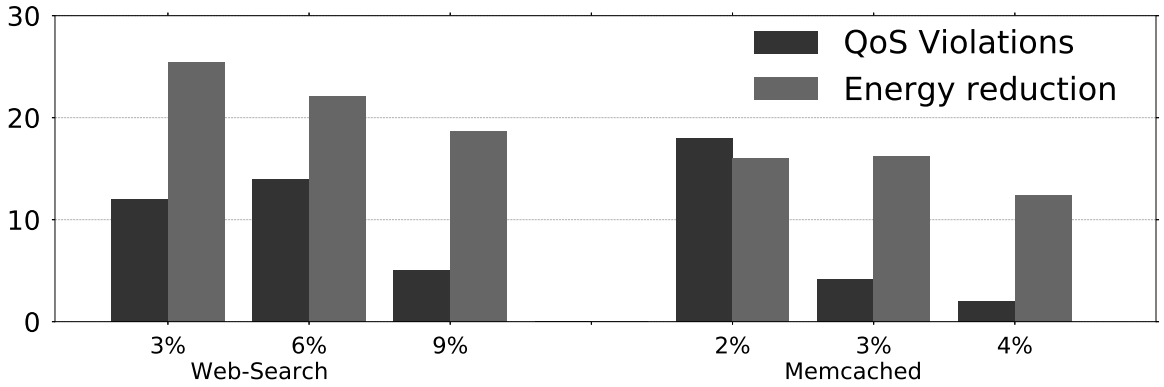


Figure 5.7 Impact of bucket size on HipsterIn QoS guarantees, and energy savings,normalized to static (all big cores) on Web-Search and Memcached.

Impact of bucket sizes — Figure 5.7 shows the impact on QoS and energy savings when varying the load bucket sizes in Hipster. The *x*-axis represents the bucket size, expressed as the percentage of maximum load. Each bar in the figure (*y*-axis) represents the QoS violations and energy reductions normalized to Static (all big cores). Using a large bucket size forces Hipster to use the same core configuration across a wide range of loads, whereas using a small bucket size allows fine-grained control. A small bucket size therefore improves the energy savings, but it tends to cause rapid changes in core configuration for small changes in load, and doing so incurs a larger number of QoS violations. On the other hand, larger bucket sizes provide better QoS guarantee but lower energy savings, because they categorize large variations in load into a single load bucket. Therefore, in tuning Hipster, we empirically determine the bucket size to maximize energy savings subject to at least 98% QoS guarantee.

5.2.3 HipsterCo Results

This section evaluates the effectiveness of HipsterCo, as a policy for collocating a single latency-critical workload and a mix of batch workloads. The objective is to maximize the throughput of the batch workloads while satisfying QoS of the interactive workloads.

Figure 5.8 shows the QoS guarantee (top), throughput (middle) and energy consumption (bottom) for Web-Search collocated with batch workloads, managed by Octopus-Man and HipsterCo. All figures are normalized to a static mapping that allocates the latency-critical workload to the two big cores and the batch workloads to the four small cores. The number of running batch workloads is equal to the number of cores not utilized by Web-Search. We report the system throughput by aggregating the IPS of all batch programs.

As shown in the top plot of Figure 5.8, HipsterCo consistently delivers 94% QoS guarantees, whereas Octopus-Man has much lower QoS guarantees of 76%. This is because Hipster

learns from the QoS behaviour and performance history and is able to jump directly to a core mapping and DVFS state that satisfies QoS. As a result, it incurs fewer core migrations compared with Octopus-Man (see Section 5.2.2.3), so it achieves superior QoS guarantees.

As shown in the middle plot of Figure 5.8, for all benchmarks, Hipster and Octopus-Man deliver much higher throughput compared to static mapping, with an average of 2.3× and 2.6× improvement, respectively. Both task managers improve performance compared with the static mapping because they migrate the latency-critical workload to small cores during periods of low load, allowing the batch workloads to run on big cores (which can be 2.6× more powerful than small cores). For *Calculix*, a compute-bound application, HipsterCo achieves the highest throughput improvement over static of 3.35×, and for *libquantum*, a memory-bound program, the least improvement is still 1.6×.

As shown in the bottom plot of Figure 5.8, HipsterCo reduces the energy consumption to an average of 80% of static, whereas Octopus-Man increases energy to an average of 1.2 times that of static. This is because, as shown in Figure **??**, Hipster explores a wider range of core configurations, including DVFS settings and mixing core types. In contrast, Octopus-Man only allows the latency-critical workload to occupy a single cluster and each cluster is set to the highest DVFS.

HipsterCo sometimes chooses a different performance–energy trade-off than Octopus-Man. An example is *lbm*, a memory bound workload, for which HipsterCo delivers 40% the throughput of Octopus-Man, but 31% lower energy. There are two main reasons for this. Firstly, when HipsterCo uses DVFS for the latency-critical workloads, this DVFS setting also applies to batch workloads running in the same cluster, reducing both batch throughput and system energy. Secondly, HipsterCo sometimes uses a larger number of cores at lower DVFS, leaving fewer resources available for the batch workloads. As a result, on average, HipsterCo marginally reduces performance (by 7%) but it delivers energy savings of 33%, both compared with Octopus-Man.

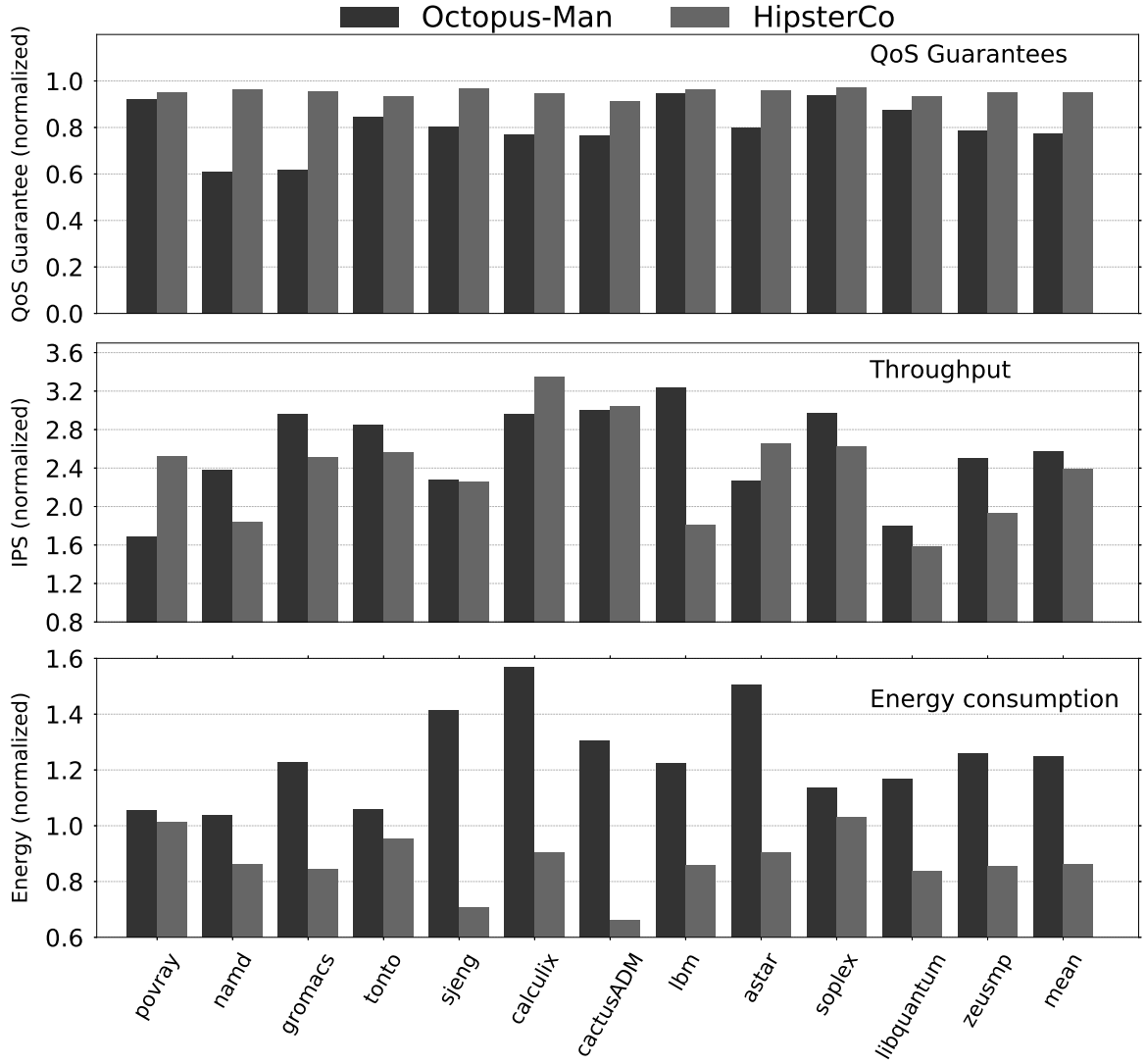


Figure 5.8 QoS guarantee (top), Throughput improvement (middle) and Energy consumption (bottom) when Web-Search is collocated with batch workloads. The results are normalized to static all big cores.

PART IV:

EPILOGUE

Errors using foo barbazinadequate data are much less than those using no data at all.

CHARLES BABBAGE

CHAPTER 6

Related Work

CHAPTER 7

Conclusion

PART V:

BIBLIOGRAPHY

Errors using foo barbazinadequate data are much less than those using no data at all.

CHARLES BABBAGE

References

[1] John Russell. ARM Unveils Scalable Vector Extension for HPC at Hot Chips. Accessed: 2016-08-09.

[2] Mont-Blanc. Mont-blanc. Accessed: 2016-08-09.

[3] Mont-Blanc 2. Mont-Blanc 2, European scalable and power efficient HPC platform based on low-power embedded technology. Accessed: 2016-08-09.

[4] Bull atos Technology. Mont-Blanc European project gains new impetus for its climb to Exascale. Accessed: 2016-08-09.

[5] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems - SIGMETRICS '12*, page 53, New York, New York, USA, 2012. ACM Press.

[7] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in 'Homogeneous' Warehouse-Scale Computers: A Performance Opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, 2 2011.

[8] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. QuickIA: Exploring heterogeneous architectures on real prototypes. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–8. IEEE, 2 2012.

[9] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76. IEEE, 3 2016.

[10] M. Guevara, B. Lubin, and B. C. Lee. Navigating heterogeneous processors with market mechanisms. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 95–106. IEEE, 2 2013.

[11] Daniel Wong and Murali Annavaram. KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–130. IEEE, 12 2012.

[12] Jason Cong and Bo Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design - ISLPED '12*, page 345, New York, New York, USA, 7 2012. ACM Press.

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

[14] AMD. *AMD64 Architecture Prog. Manual Volume 2: System Prog.* AMD Corp.

[15] ARM. *Infocenter for ARM Cortex-A57*. ARM.

[16] Applied Micro XGene2. http://goo.gl/XA04r1, 2016. Online; accessed 31 August 2016.

[17] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *2011 International Green Computing Conference and Workshops*, IGCC 2011, pages 1–8, July 2011.

[18] H. Peter Anvin. MSR tools. Accessed: 2014-10-14.

[19] Wattsup Pro. https://www.wattsupmeters.com/secure/products.php, 2016. Online; accessed 31 August 2016.

[20] ARM Juno Power. http://goo.gl/JryrgT, 2016. Online; accessed 31 August 2016.

[21] ARM. Arm juno power registers, 2016. Accessed: 2016-6-27.

[22] ARM. SYS_POW_SYS Register, https://goo.gl/fmTTQi.

[23] Linux. Perf: Linux profiling with performance counters.

[24] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 30(4):65–79, 7 2010.

[25] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 9 2014.

[26] S.Eranian. Issues with libpfm4.7.0 and perf on arm juno r0, 2016. Accessed: 2016-6-27. fix

[27] ARM Limited. ARM ® Cortex ® -A53 MPCore Processor Technical Reference Manual.

[28] ARM Limited. ARM ® Cortex ® -A57 MPCore Processor Revision: r1p0 Technical Reference Manual.

[29] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 42(3):301–312, 10 2014.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles. *ACM SIGARCH Computer Architecture News*, 43(3):450–462, 6 2015.

[31] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik. In *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pages 598–610, New York, New York, USA, 12 2015. ACM Press.

[32] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 7 2013.

[33] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258. IEEE, 2 2015.

[34] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. *SIGARCH Comput. Archit. News*, 39(3):319–330, June 2011.

[35] Jason Mars and Lingjia Tang. Whare-map. *ACM SIGARCH Computer Architecture News*, 41(3):619, 7 2013.

[36] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux. *ACM SIGARCH Computer Architecture News*, 41(3):607, 7 2013.

[37] Tribuvan Kumar Prakash and Lu Peng. Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor. In *Proceedings of the ISAST Transactions on Computers and Software Engineering*, ISAST 2008, pages 36 –41. IEEE, 2008.

[38] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 9 2006.

[39] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[40] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A reiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *22nd International Symposium on Computer Architecture*, ISCA 1995, pages 24–36, June 1995.

[42] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[43] Memcached. http://memcached.org, 2016. Online; accessed 31 August 2016.

[44] Elasticsearch. https://github.com/elastic/elasticsearch, 2016. Online; accessed 31 August 2016.

[45] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In

Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[46] L.A. Barroso, J. Dea bad nd U. Holzle. Web search for a planet: the google cluster architecture. *IEEE Micro*, 23(2):22–28, 3 2003.

[47] Faban. https://faban.org, 2016. Online; accessed 31 August 2016.

[48] Michael Ferdman, Babak Falsafi, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, and Anastasia Ailamaki. Clearing the clouds. *ACM SIGARCH Computer Architecture News*, 40(1):37, 4 2012.

[49] Rajiv Nishtala, Daniel Mossé, and Vinicius Petrucci. Energy-aware thread co-location in heterogeneous multicore processors. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 21:1–21:9, Piscataway, NJ, USA, 2013. IEEE Press.

[50] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[51] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[52] J. Mars, Lingjia Tang, and R. Hundt. Heterogeneity in homogeneous; warehouse-scale computers: A performance opportunity. *Computer Architecture Letters*, 10(2):29–32, July 2011.

[53] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Intel. Intel intelligent power node manager.

[55] Openstack. Openstack.

[56] Tivoli Management Framework. Intel intelligent power node manager.

[57] Sergey Blagodurov. *Addressing Shared Resource Contention in Datacenter Servers*. PhD thesis, Simon Fraser University, August 2013.

[58] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, July 2009.

[59] A. Leonard Brown. The State of ACPI in the Linux Kernel. *SIGARCH Comput. Archit. News*, 1(1):121–132, 2004.

[60] Rajiv Nishtala, Marc González Tallada, and Xavier Martorell. A methodology to build models and predict performance-power in cmps. In *44th International Conference on Parallel Processing Workshops, ICPPW 2015, Beijing, China, September 1-4, 2015*, pages 193–202, 2015.

[61] Rajiv Nishtala, Xavier Martorell, Vinicius Petrucci, and Daniel Mosse. REPP-H: Runtime Estimation of Performance-Power on Heterogeneous Data Centers. In *Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '16. IEEE Press, 2016.

[62] Rajiv Nishtala and Xavier Martorell. REPP-C: Runtime Estimation of Performance-Power with Workload Consolidation in CMPs. In *Proceedings of the 7th International Green and Sustainable Computing Conference*, IGSC '16. IEEE Press, 2016.

[63] Frank Bellosa. The Benefits of Event: Driven Energy Accounting in Power-sensitive Systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, pages 37–42, New York, NY, USA, 2000. ACM.

[64] Adam Lewis, Jim Simon, and Nian-Feng Tzeng. Chaotic Attractor Prediction for Server Run-time Energy Consumption. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[65] Canturk Isci and Margaret Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[66] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. A Systematic Methodology to Generate Decomposable and Responsive Power Models for CMPs. *IEEE Transactions on Computers*, 62(7):1289–1302, 2013.

[67] T.E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, Nov 2011.

[68] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[69] T. M. Le. A study on Linux Kernel Scheduler Version 2.6.32.

[70] Kishore Kumar Pusukuri, David Vengerov, and Alexandra Fedorova. A methodology for developing simple and robust power models using performance monitoring events. In *Proceedings of the Workshop Interaction between Operating Systems and Computer Architecture*, WIOSCA 2009, Washington, DC, USA, 2009. IEEE Computer Society.

[71] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. In *Proceedings of the 47th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, MICRO-47, pages 445–457. IEEE Computer Society, 2014.

[72] William Lloyd Bircher and Lizy K. John. Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the IEEE/ACM International Symposium on Performance Analysis of Systems Software*, ISPASS 2007, pages 158–168, Washington, DC, USA, 2007. IEEE Computer Society.

[73] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[74] Dimitrios Chasapis and Marc Casas, et al. ParsecSs: Evaluating the Impact of Task Parallelism in the Parsec Benchmark Suite. In *ACM TACO*, 2015.

[75] Hui Kang and Jennifer L. Wong. To hardware prefetch or not to prefetch?: A virtualized environment study and core binding approach. *SIGPLAN Not.*, 48(4):357–368, March 2013.

[76] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.

[77] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power Capping: A Prelude to Power Shifting. *Cluster Computing*, 11(2):183–195, June 2008.

[78] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the Effectiveness of Model-based Power Characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[79] S. Imamura, H. Sasaki, K. Inoue, and D.S. Nikolopoulos. Power-capped DVFS and thread allocation with ANN models on modern NUMA systems. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 324–331, Oct 2014.

[80] Michela Becchi and Patrick Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.

[81] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A. Kozuch. Optimality Analysis of Energy-performance Trade-off for Server Farm Management. *Perform. Eval.*, 67(11):1155–1171, November 2010.

[82] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[83] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal Power Allocation in Server Farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[84] M.A. Haque, H. Aydin, and Dakai Zhu. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[85] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *Proc. of the 44th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, MICRO-44, pages 175–185. ACM, 2011.

[86] Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. Efficient Interaction Between OS and Architecture in Heterogeneous Platforms. *SIGOPS Oper. Syst. Rev.*, 45(1):62–72, February 2011.

[87] Ozlem Bilgir, Margaret Martonosi, Qiang Wu, and Facebook Inc. Exploring the Potential of CMP Core Count Management on Data Center Energy Savings. In *Proceedings of the Workshop on Energy Efficient Design (WEED)*, June 2011.

[88] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T N Vijaykumar. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-48*, Waikiki, Hawaii, 2015. ACM Press, 2015.

[89] Haishan Zhu and Mattan Erez. Dirigent. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*, volume 51, pages 33–47, New York, New York, USA, 2016. ACM Press.

[90] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 6 2014.

[91] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: transparently identifying and managing performance interference in virtualized environments. pages 219–230, 6 2013.

[92] Qiang Wu. Making Facebook's software infrastructure more energy efficient with Autoscale, https://goo.gl/vJi1kf.

[93] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2 2015.

[94] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *2006 IEEE International Conference on Autonomic Computing*, pages 65–73. IEEE, 2006.

[95] IBM Research | Technical Paper Search | Model-Based and Model-Free Approaches to Autonomic Resource Allocation(Search Reports), 2 2007.

[96] Gerald Tesauro. Online Resource Allocation Using Decompositional Reinforcement Learning. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 886–891, 1 2005.

[97] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control. *IEEE Transactions on Computers*, 56(4):444–458, 4 2007.

[98] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail. In *Proceedings of the ACM Symposium on Cloud Computing - SOCC '14*, pages 1–14, New York, New York, USA, 2014. ACM Press.

[99] Suton. R.S and A.G Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[100] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power Management of Datacenter Workloads Using Per-Core Power Gating. *IEEE Computer Architecture Letters*, 8(2):48–51, 2 2009.

[101] Niti Madan, Alper Buyuktosunoglu, Pradip Bose, and Murali Annavaram. A case for guarded power gating for multi-core processors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 291–300. IEEE, 2 2011.