PATMOS 2018

Low-Complexity Run-time Management of Concurrent Workloads for Energy-Efficient Multi-Core Systems

Ali Aalsaud[1, 3], Ashur Rafiev[2], Fie Xia[1], Rishad Shafik[1] Alex Yakovlev[1]

[1] School of Engineering, [2] School of CS, Newcastle University, Newcastle upon Tyne, UK
[3] School of Engineering, Al-Mustansiriya University, Baghdad, Iraq

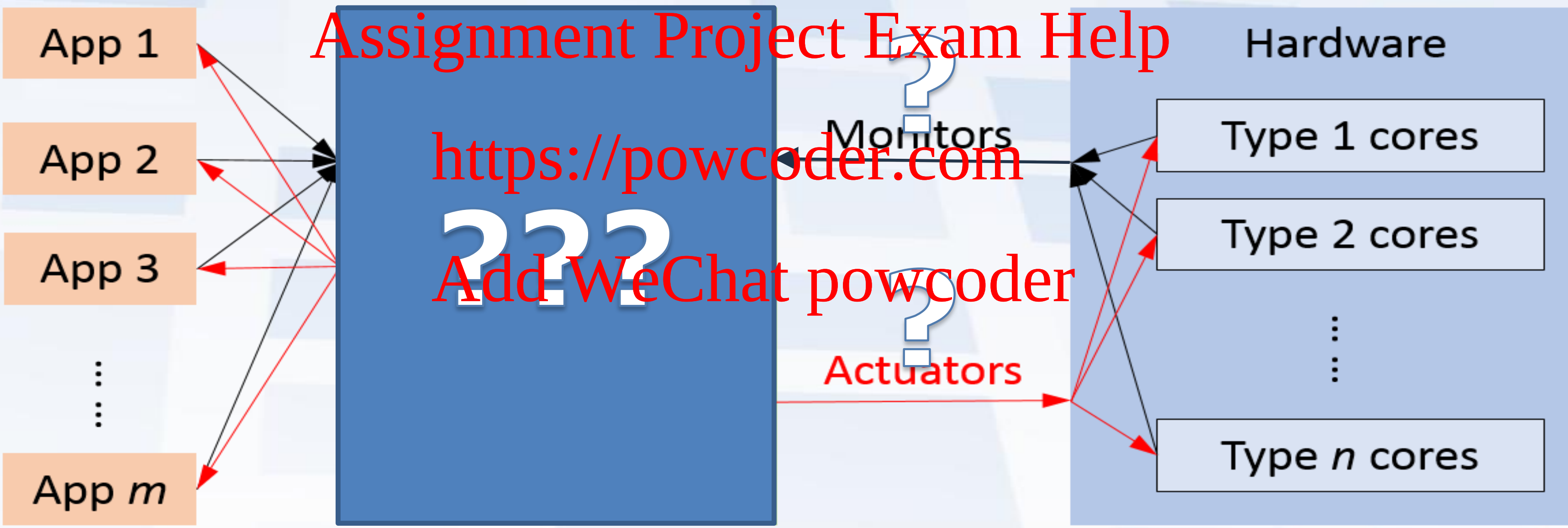
The research environment

- Hardware platforms are becoming heterogeneous multi-/many-core
- Execution is increasingly multiple applications running concurrently
- Energy/power is becoming a major concern
  - Limited by both local and global issues
- When mapping multiple concurrent applications onto heterogeneous many-core, it is possible to optimize performance, energy, or both
- Hardware may support the selection of cores and the setting of core voltages and frequencies
- Apps may be run on single or multiple cores

Newcastle University

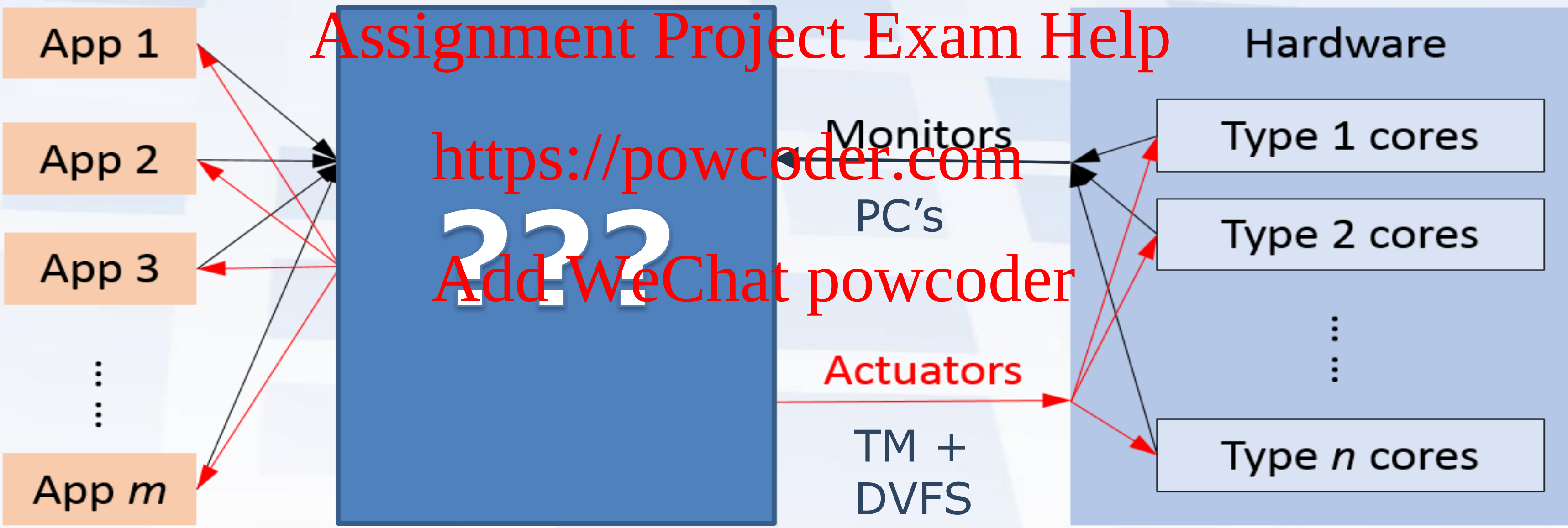
PRiME www.prime-project.org

Our problem



A control method that optimizes some metric by tuning the execution

Monitors and actuators decision



Performance counters for monitoring
Thread-to-core mapping (TM) and DVFS as actuators

The RTM complexity

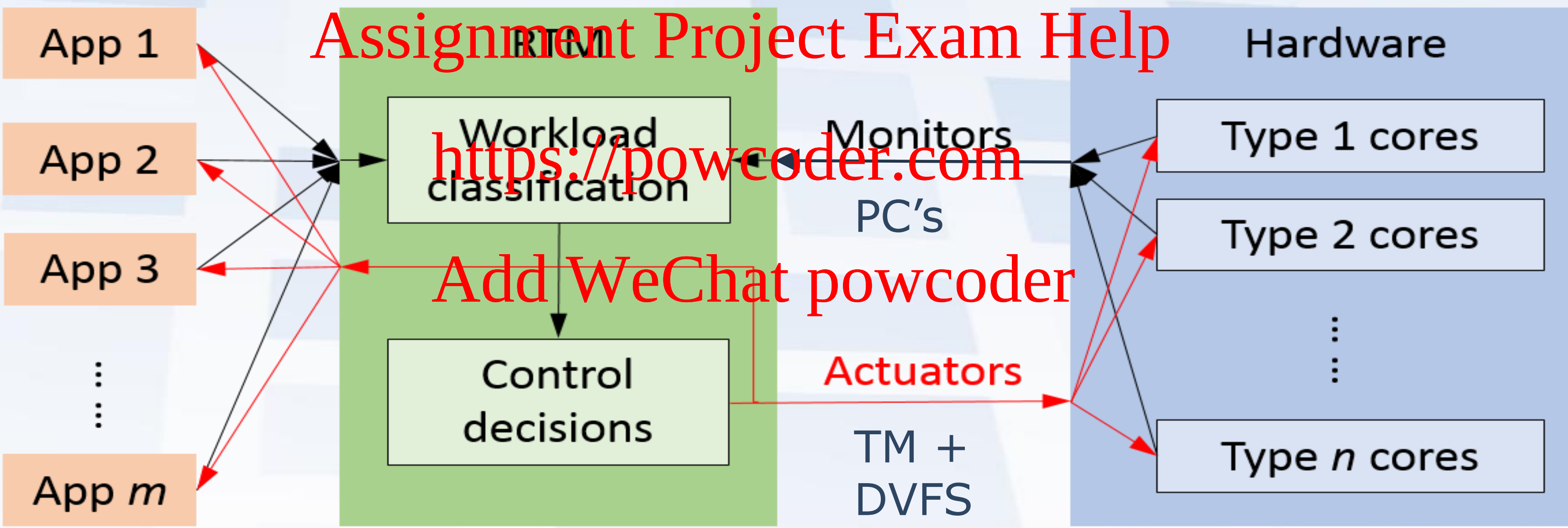
- The size/complexity of the decision space
  - Number of cores, DVFS points, types of cores, etc are combinatorial
  - Then it is exponential with the number of applications running in parallel
  - Example with a 7-core system
    - With max 7 apps

N_{apps}	brute force	valid
1	20	19
2	400	111
3	8000	309
4	1.6×10^5	471
5	3.2×10^6	405
6	6.4×10^7	185
7	1.28×10^9	35

Runtime management (RTM)

- Existing approaches deal with (or avoid) this problem in a variety of ways
  - Homogeneous platforms of limited numbers of cores
    - OK up to several years ago
  - Single application scenarios
    - OK for embedded systems designed to do one thing
  - TM or DVFS, but not both, or one of them offline and the other runtime
  - Offline (static) design of the RTM
    - Reduces the runtime workload of the controller itself ☺
    - Runtime variations of applications cannot be modelled easily ☹
  - Workload classification (WLC) – organize applications into types (classes)
    - Reduces the design/execution decision space of the RTM ☺
    - Taxonomy needs to be supported by good arguments

RTM design

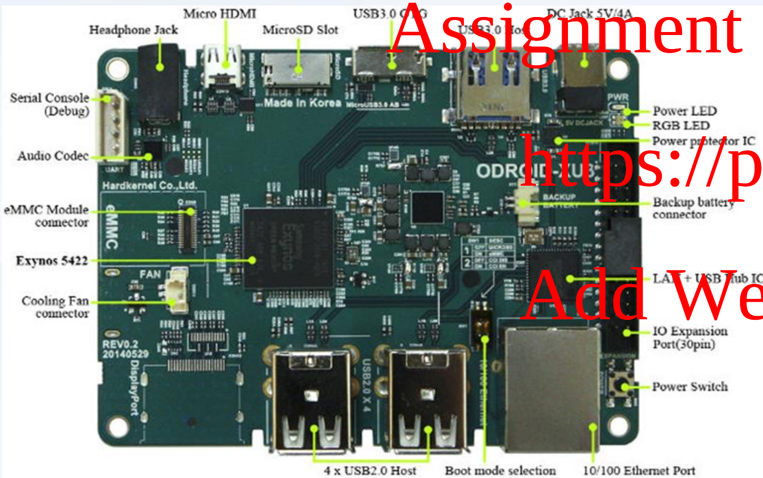

App 1
App 2
App 3
App m
RTM
Workload classification
Control decisions
Monitors
PC's
Actuators
TM +
DVFS
Hardware
Type 1 cores
Type 2 cores
Type n cores

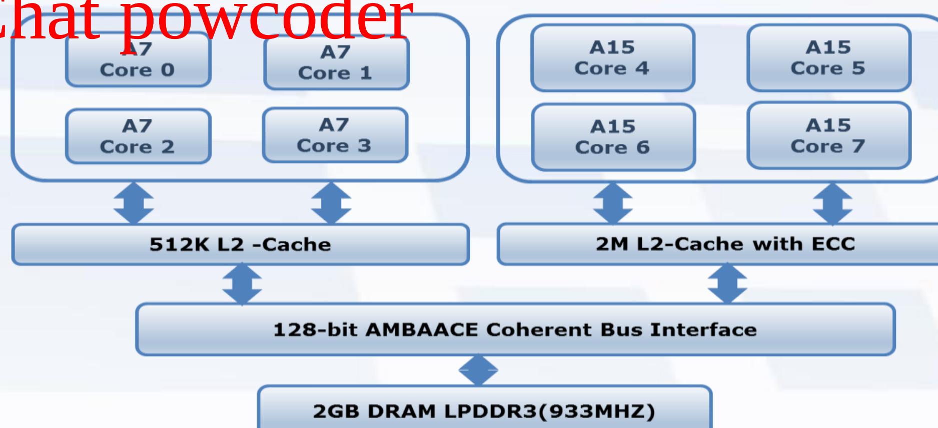
WLC reduces the decision space

- But what classification taxonomy and which classifiers?
- The hypothesis (reasonably common) is that memory usage is a good classifier (for platforms based on shared-memory)
  - For performance and power, memory-heavy threads need different DVFS and TM decisions from CPU-heavy ones
- We validated this hypothesis with a large number of experiments
  - Running our own synthetic benchmark, whose memory use rate can be directly tuned, on a number of system platforms including homogenous and heterogeneous multi-core
  - Memory-heavy and CPU-heavy threads always demand different TM and DVFS decisions to yield ‘the best’ for our chosen metric
  - We chose power-normalized performance (PNP) but this should be true for any metric which combines both performance and power (e.g. EDP)

Experimental platform

- Odroid XU3 based on ARM big.LITTLE architecture





WLC RTM design how to, v. 1

- Find existing benchmark applications, from e.g. Parsec
  - Many of them have memory-heavy, cpu-heavy labels attached
- Characterize the h/w platform with these apps
  - Find out the optimal TM+DVFS for combinations of memory-heavy, cpu-heavy apps in concurrent execution
  - Build RTM algorithm
- Existing benchmark apps are a poor choice for characterization and offline optimization
  - They have phases of cpu-heavy and memory-heavy so characterization results are not trustworthy ☹

PRiME www.prime-project.org

Newcastle University

Our synthetic benchmark

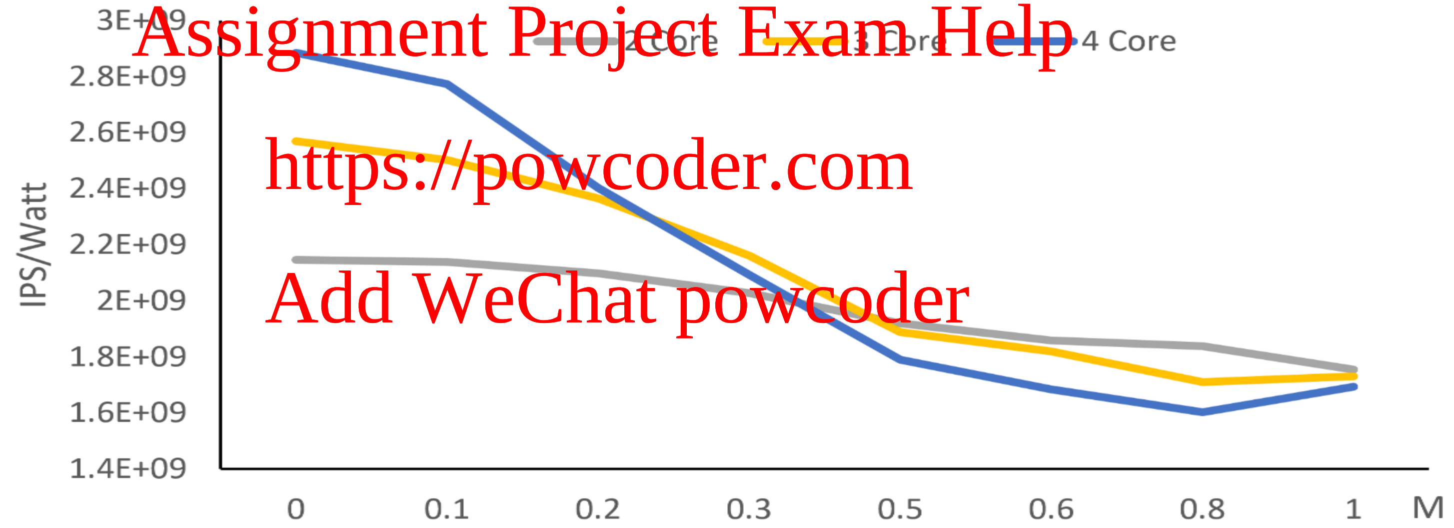
- It tunes the ratio of memory-access instructions among all instructions at the lowest granularity for a high-level programming language (M=memory/memory+CPU)
- For high-level languages, tuning M between 0 and 1 cannot produce pure CPU or pure memory-access code ☹
  - But this is linear to the true underlying memory-access ratio where its range does cover ☺
  - This is no worse than if you asked app developers to instrument their memory usage rate via a special API 😐
  - The hypothesis is that if you carefully constructed this type of synthetic benchmark in machine code, you may be able to have full(er) cover of the memory-access rate

PRiME www.prime-project.org

Newcastle University

Synthetic benchmark

- Some experimental results



- We use this benchmark to develop RTM decisions

WLC taxonomy

- *Class 0*: low-activity workloads
- *Class 1*: CPU-intensive workloads
- *Class 2*: CPU- and memory-intensive workloads
- *Class 3*: memory-intensive workloads
- Contingency classes
  - *Class u*: unknown class (being classified)
  - *Class lp*: low parallelizability

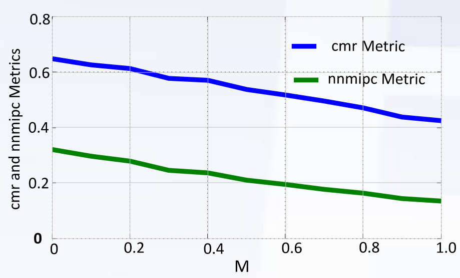
Classification-based RTM

- Lookup table approach

Metrics	Definitions
nipc	$(InstRet/Cycles)(1/IPC_{max})$
iprc	$InstRet/ClockRef$
nnmipc	$(InstRet/Cycles – Mem/Cycles)(1/IPC_{max})$
cmr	$(InstRet – Mem)/InstRet$
uur	$Cycles/ClockRef$

Class	frequency	A7	A15
0	min	single	none
1	max	none	max
2	min	max	max
3	max	max	none
u, lp	min	none	single

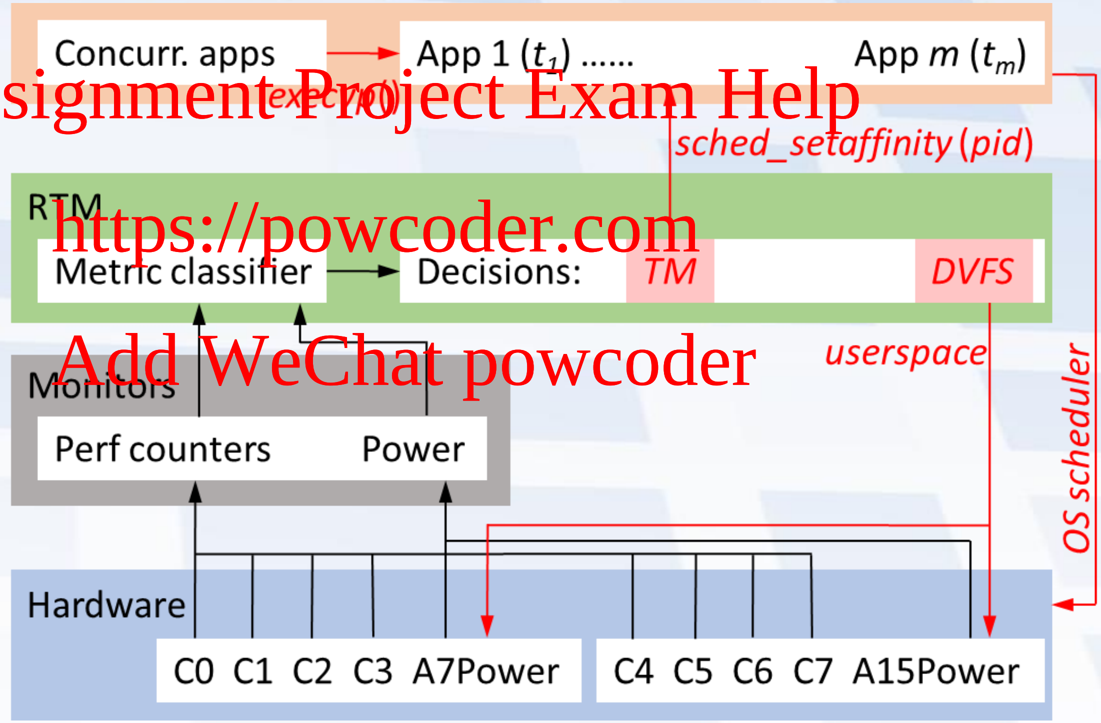
By running the *M* benchmark



Metric ranges	Class
urr of all cores [0, 0.11]	0: low-activity
nnmipc per-core [0.35, 1]	1: CPU-intensive
nnmipc per-core [0.25, 0.35)	2: CPU+memory
nnmipc per-core [0, 0.25)	3: memory-intensive

Classification-based RTM

RTM within the RTMx environment



Alternative WLC method

- Each app has a class label attached
  - Could be obtained through offline static characterization
  - Could be attached by the app-creators (programmers)
    - Annotations in code
    - Workload instrumentation facilities in code
    - May oblige these through a classification-friendly API
- The idea that each app has a class throughout its execution is a false assumption
  - This makes all of the above rather non-optimal

Newcastle University

Moving from offline to RT

- Workloads belong to different classes in different execution phases
  - Could be CPU-intensive then memory-intensive then low-activity, etc.
  - Complex behaviours possible
  - RTM has per-interval (re)classification
  - No off-line static classification
  - No app annotation
  - No app instrumentation

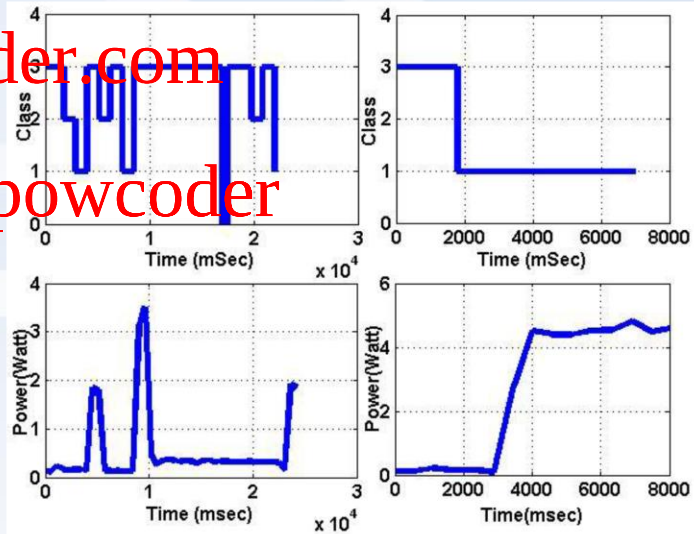
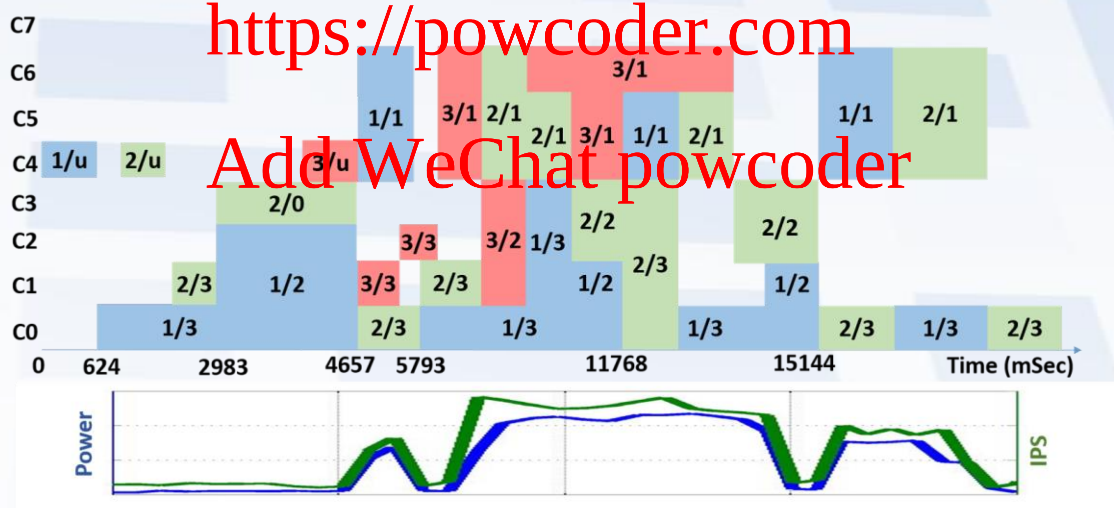


Figure 6 *Fluidanimate* (left) and *ferret* (right) classification and power traces.

An execution trace

- C7 is not used – held in reserve for classifying any new joining app if C4-6 are occupied
- Power and throughput closely track the number of A15s in use
- Apps (Parsec and synthetic benchmarks) have dynamic classes and get allocated different type and number of cores



Classification-based RTM

- Difficult to prove optimality for this approach ☹
- Complexity is very low (overhead essentially zero) ☺

$$O(N_{app} * N_{class} + N_{core})$$

- Results are encouraging

Application Scenarios	Workload Classification (WLC)	Multivariate Linear Regression (MLR)	MLR + WLC
Fluidanimate alone	127%	127%	139%
Two different class applications	68.60%	61.74%	128.42%
Three different class applications	46.60%	29.30%	61.27%
Two Class 3 applications	24.50%	19.81%	40.33%
Three Class 3 applications	44.40%	36.40%	58.25%
Two Class 1 applications	31.00%	26.53%	41.74%

– Improvements over *Ondemand*

Issues

- Possible to have classification oscillations
  - Control cycles coinciding with app phase change points, rare
    - Easily solved by reducing control cycles
  - App on the boundary of classification threshold, rare
    - Easily solved by boundary tuning and/or increasing the number of classes for a higher class resolution
  - App is not parallelizable
    - Given one core, it is CPU-intensive; then give it lots of cores, it becomes low activity; then give it one core; etc.
    - Detectable after two cycles and re-classify as low-paralellizability for a single large core
  - Nyquist/Shannon sampling requirement likely plays a role

Classification-based RTM

- Method recap
  - Decide on a classification scheme based on system architectures and optimization targets
  - Generate synthetic benchmarks that allow the tuning of classifiers
  - Run these benchmarks on the target platforms to generate
    1. classification thresholds to build the classification table
    2. control decisions to build the control table
  - If possible, validate with real expected applications to find potential problems with the RTM
  - Fine-tune the RTM, looping back to earlier steps may be needed

PRiME www.prime-project.org

Newcastle University

PRiME
www.prime-project.org

Newcastle University

ROMA 2020

Thanks

www.async.org.uk