

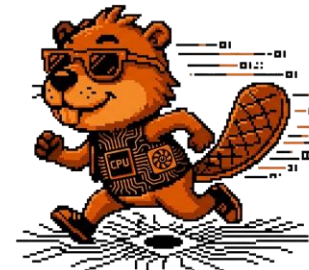
MEMPOWER: Efficient **Power Management** with Fine-grained **Memory Analysis** and **Modeling** for HPC Workloads

Nanda Velugoti[†], Joseph Manzano[‡], Andres Marquez[‡],
Nathan Tallent[‡], Kyle Hale[†]

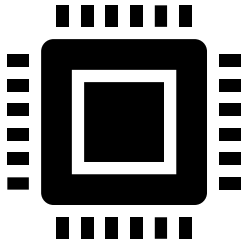


Oregon State[†]
University

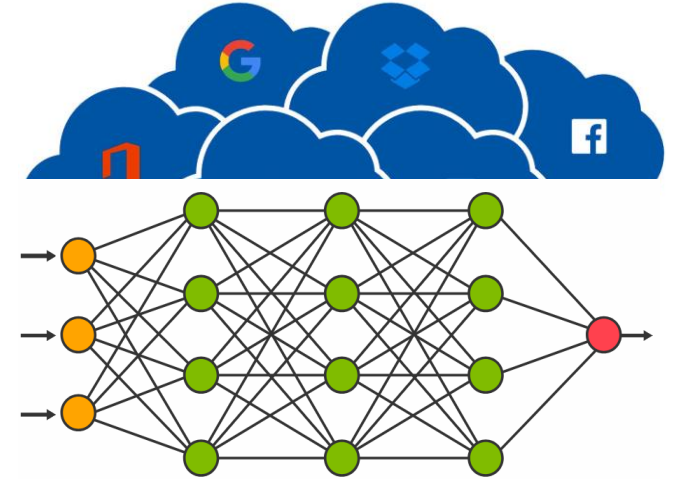
Pacific Northwest[‡]
NATIONAL LABORATORY



Importance of managing power for CPUs



CPU



More Compute

More Power

Efficient Mgmt.

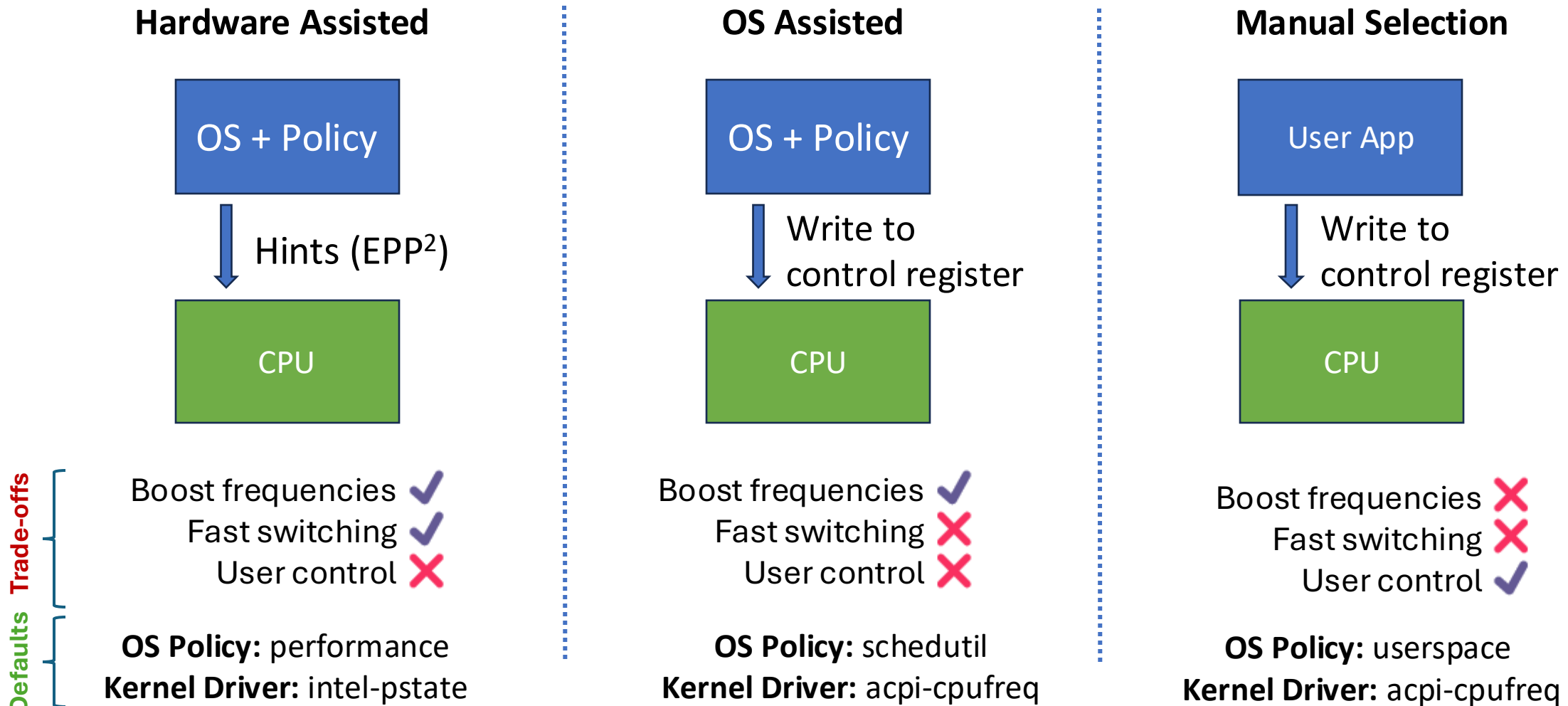
Key questions:

- Are current default power schemes inefficient?
- Can we do better?
- **Yes and yes**

I will cover **how & why**,

- default power mgmt. miss out on key memory behavior info
- MEMPOWER capitalizes on it and outperforms them (**1.72x EDP↓**)

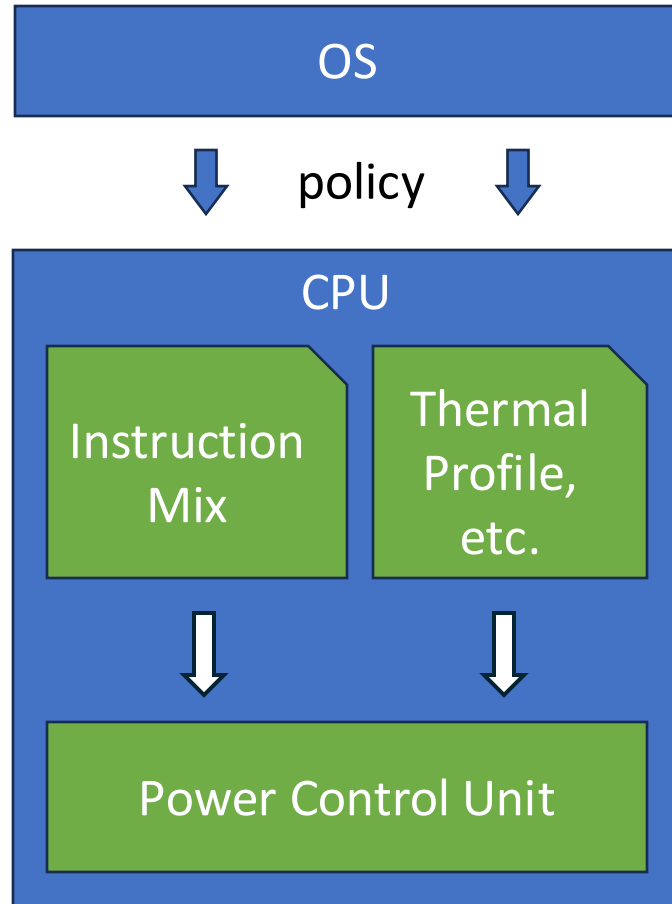
Intro to OS/CPU¹ power management (DVFS)



1. We focus on Linux OS & Intel CPUs

2. EPP = Energy-Performance Preference

CPU power management process



Process:

- OS sets its policy for CPU
 - CPU samples instruction mix over time windows
 - scalar integer
 - floating-point
 - vector/AVX
 - loads/stores
 - branches
- Relative amounts each of instruction type in a window → Instruction Mix
- PCU takes the instruction mix + other perf counters
 - Finally, PCU chooses a frequency-voltage state (P-state)

So, what's missing?

- The instruction mix **does not** capture fine-grained memory behavior
- Especially, can't identify **Regular vs. Irregular** accesses

Why does this matter?

Regular vs. Irregular Access

Regular/Strided Accesses

- k -stride iteration ($k=1,2,\dots$)
- Matrix Multiplication

Irregular Accesses

- Pointer chasing
- Sparse Matrix Operations

What are the effects on CPU optimizations?

- ✓ Good for caching
- ✓ Good for prefetching
- ✓ Less CPU stalls

- ⚠ Bad for caching
- ⚠ Bad for prefetching
- ⚠ More CPU stalls

Irregular accesses → Cache misses → CPU is waiting for DRAM!

Why does this matter in the context of power setting?

Lets see an example...

Case study: miniVite

miniVite:

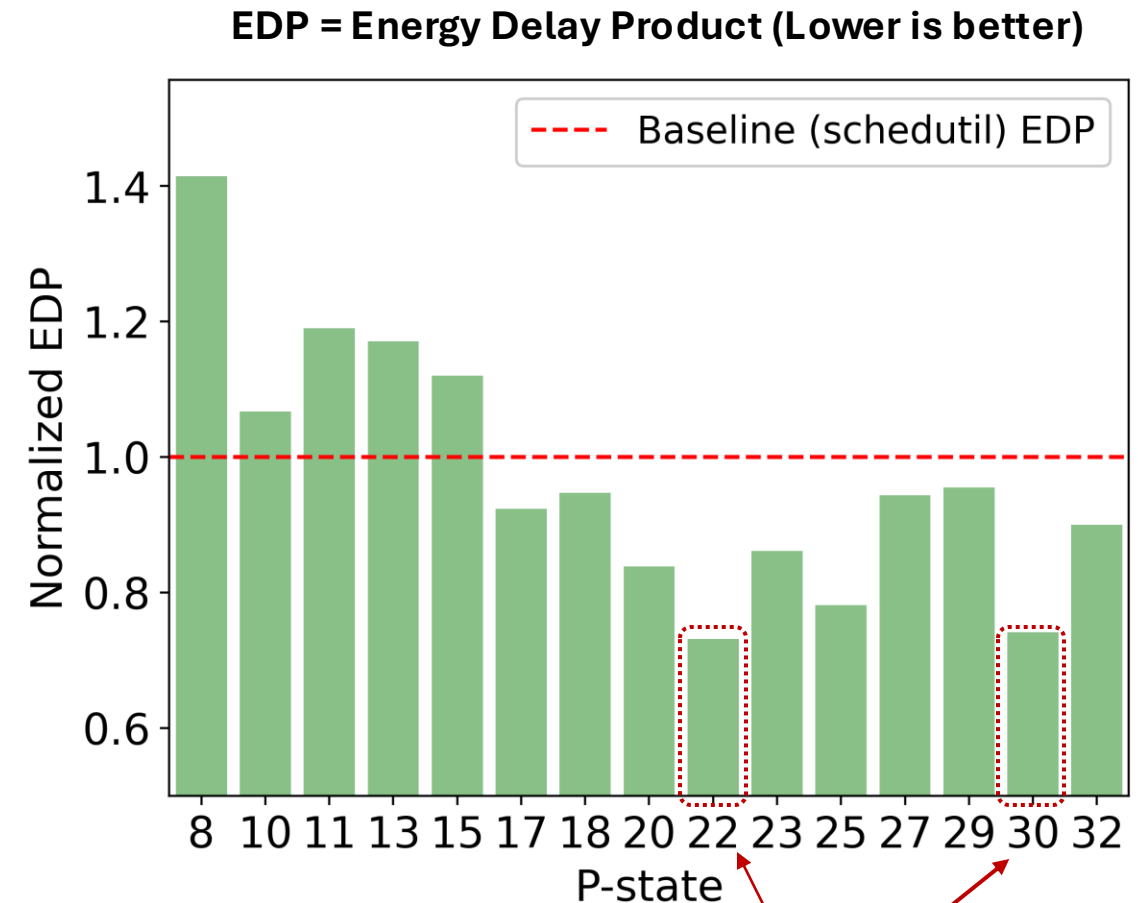
- parallel graph community detection
- mem-intense irregular code regions

Experiment:

Run every irregular memory intense code region at ***all*** available P-states manually.

How does OS default perform compared to manual selection?

Takeaway 1: OS default does not capitalize on irregular memory access regions



**Static P-state settings
with better EDP**

Limitations of default power management

Q: Why can't the OS/hardware defaults be better?

- Has performance counters: cache-misses
 - Can infer irregularity at a high-level, but not enough
 - Instruction mix is not fine-tuned to a particular application
- **Problem:** They can not capture fine-grained memory access behavior
 - i.e., **individual memory accesses over time**

Solution:

- Use a low-cost profiler to capture access trace (MemGaze¹)
- Analyze the trace to find mem-intense code regions

Characterizing irregular mem-intensity

Does CPU
Stall?

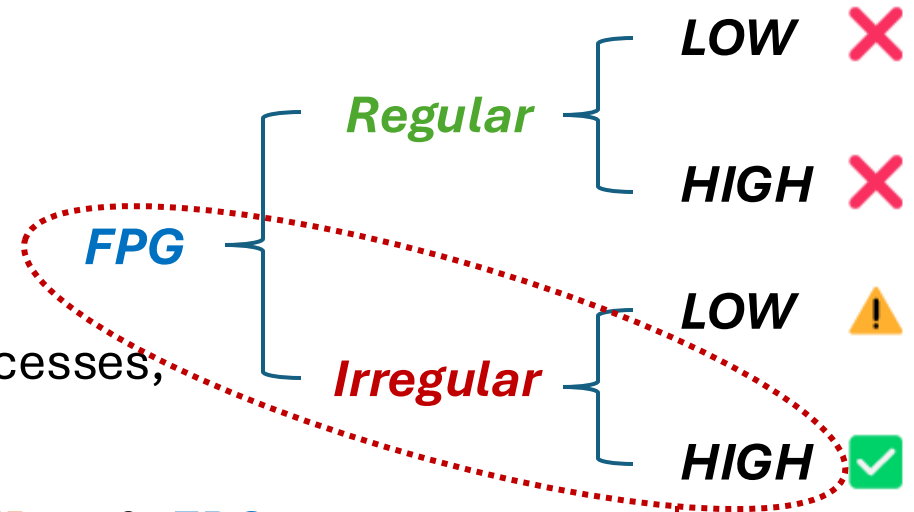
Footprint (FP): Number of total unique accesses

- *abbacbaed* → *abcde* → **FP: 5**

Footprint Growth (FPG): Rate of change of **FP** w.r.t accesses,

$$FPG = (FP_{new} - FP_{old}) / FP_{old}$$

- **T0:** *abbacbaed* → *abcde* → **FP: 5** & **FPG: ∞**
- **T2:** *abbacbaedaefb* → *abcdef* → **FP: 6** & **FPG: $1/5 = 0.2$**
- **T5:** *abbacbaedaefbgafhc* → *abcdefgh* → **FP: 8** & **FPG: $2/8 = 0.25$**



High Irregular FPG → Cache misses → CPU is waiting for DRAM!

Can we do better than OS default?

💡 **Idea:** Instead of searching for ideal P-states exhaustively, we can utilize an Irregular FPG!

If **Irreg. FPG** ↑ then,
access latency is ↑
(vice-versa)



If **Irreg. FPG** ↑ then,
P-state can be ↓
(vice-versa)

Takeaway 2: Use Irreg. FPG to identify irregular code regions and select a better P-state

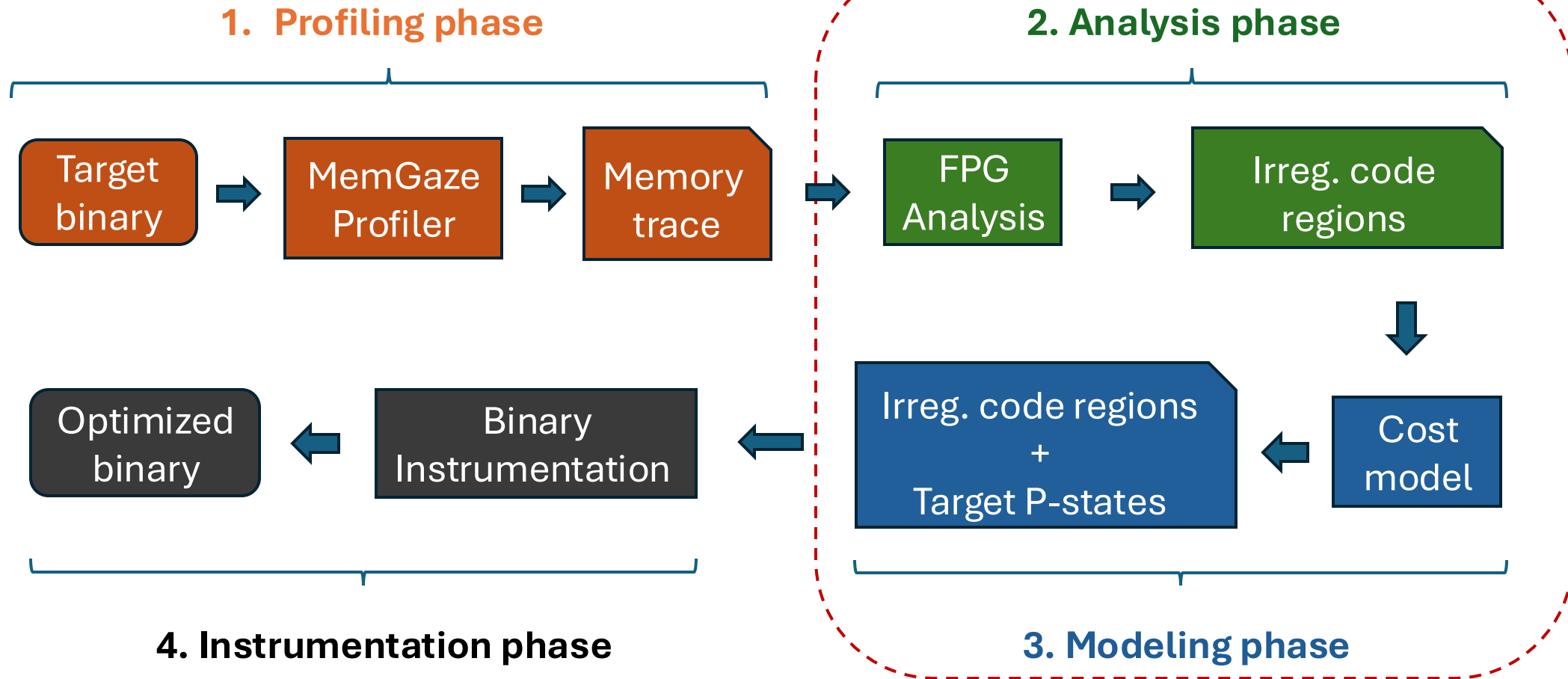
We propose MEMPOWER: A framework to,

1. capture Irreg. FPG for all code regions of a target app and,
2. manually choose a beneficial P-state based on Irreg. FPG

Lets see MEMPOWER in detail ...

MEMPOWER pipeline

Phases 2 & 3 are covered in his talk.
For more details, please refer to the paper.



Extracting mem-intense code regions

Captured memory trace:

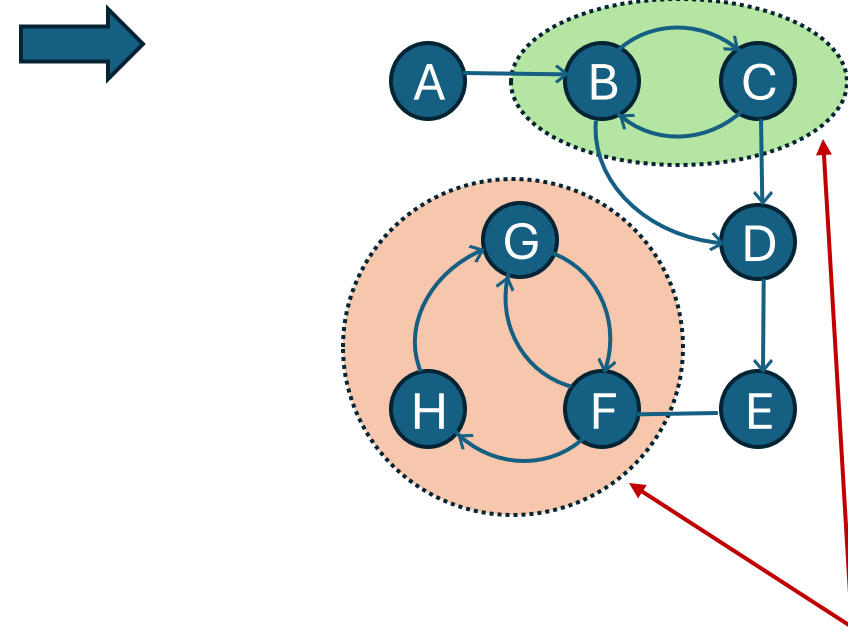
Instruct. Pointer	Basic Block	Memory Address	Time (ns)	Access Type
0xdeadbeef	A	0xn00bfac	1234.4321	strided
...	...	...	...	...

Access Type: constant / strided / irregular

Code regions attributed with FPG metrics:

Code Region	FPG _{irregular}	FPG _{strided}	FPG _{constant}
CR0	0.62	0.41	0.00
...	...	...	...

Basic block transition graph:



Clusters = hot irreg. memory sequences over time, aka., Mem-intense Code Regions

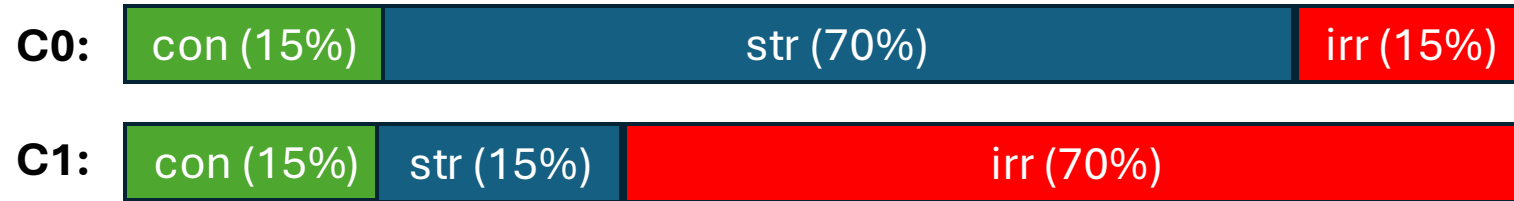
Quantifying irregularity

Given access types:

- Constant
- Strided
- Irregular

Problem: for a given code region, how can we quantify the amount of irregularity?

For example, consider two code regions C0 & C1:



Both **C0** and **C1** have irregular accesses,
but only **C1** is worth running at a lower P-state

Solution: compute relative irregular FPG for each code region using the formula,

$$\text{Relative } F_{\text{irr}} = \frac{F_{\text{irr}}}{F_{\text{irr}} + F_{\text{str}} + F_{\text{con}}}$$

How to choose P-states?

How about we choose ideal (empirical) P-states for each code region?

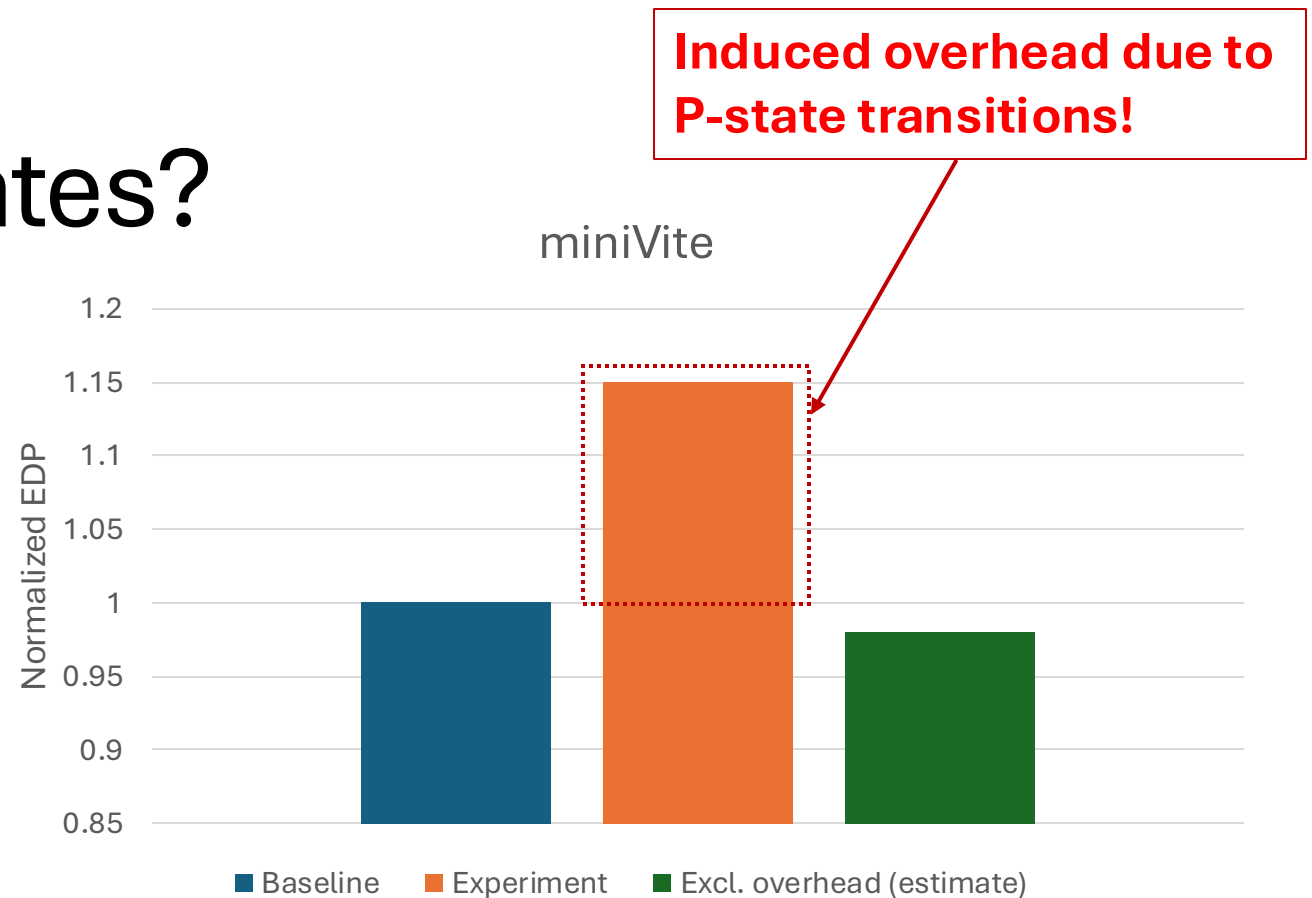
Experiment: For miniVite,

1. find the best P-state for each region individually (via separate runs)
2. run the whole app with the best P-state per code region

We expect the performance to be better than baseline (OS default).

What do we see?

Takeaway 3: P-state transition cost needs to be amortized over time



$$C = \frac{P \times L}{T \times F_{\text{irr}}}$$

Cost (C) ↓ P-state ↓
(vice-versa)

- P-state transition overhead (**P**)
- Level in CCT* (**L**)
- Total Elapsed Time (**T**)
- Irreg. FPG (**F_{irr}**)

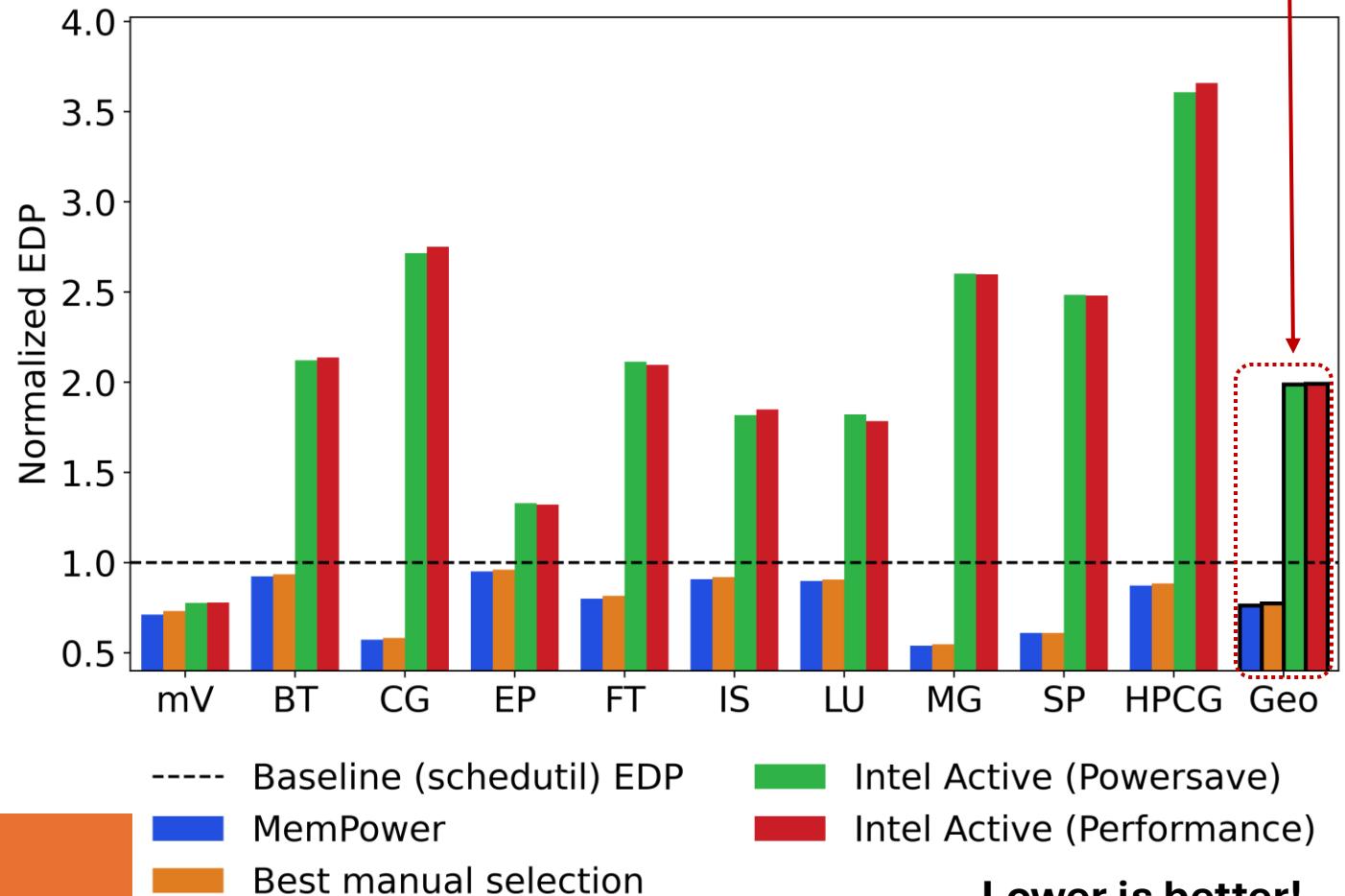
How does MEMPOWER perform?

MEMPOWER improves overall EDP by 1.25x

Workloads:

- miniVite
 - Input: arabic-2005 graph
- NAS Parallel Benchmarks
 - Input: Class B
- HPCG
 - Input:
 - nx=104
 - ny=104
 - nz=104
 - rt=6

Average Execution Time = ~60-70s



Takeaway 4: MEMPOWER outperforms SOTA defaults

Related Work

OS

- PIE [ISCA '12]
- POW [HPDC '15]
- PShifter [HPDC '18]

Compiler

- DVS Algo [PLDI '03]
- DAE [ICS '13]
- PULSE [ASPLOS '25]

Library/Runtime (MPI)

- PART [SC '05]
- Adiago [SC '07]
- COUNTDOWN [ANDARE '18]

Application

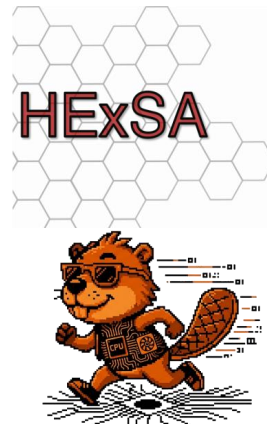
- Profiled DVFS [ISCA '03]
- PowerDail [ASPLOS '11]
- Turbo Diaries [USENIX ATC '14]

MEMPOWER

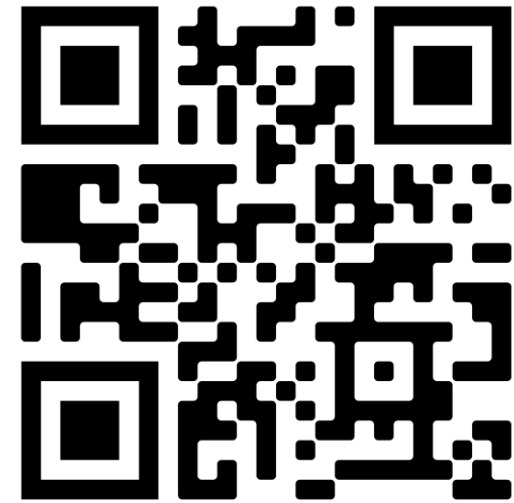
- ✓ No online analysis
- ✓ Low cost profiling
- ✓ No simulation
- ✓ App agnostic
- ✓ No code change
- ✓ Extensible

Contributions

1. Show OS/hardware does not capitalize on irregular memory accesses
2. MEMPOWER uses Irregular FPG to find memory intensive regions
3. MEMPOWER employs a cost model to find beneficial P-state transitions
4. MEMPOWER outperforms OS/hardware power defaults



**For more details
please read our paper:**



SCAN ME

**Thank you!
*Any questions?***