Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand

Amin Ansari[1], Shuguang Feng[2],
Shantanu Gupta[3], Josep Torrellas[1], and Scott Mahlke[4]

[1] University of Illinois, Urbana-Champaign
[2] Northrop Grumman Corp.
[3] Intel Corp.
[4] University of Michigan, Ann Arbor

HPCA-19

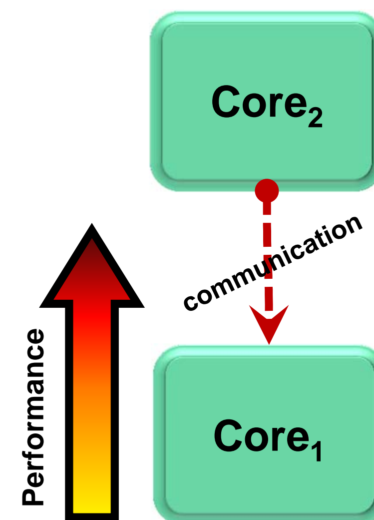
February 27, 2013

Adapting to Application Demands

- Number of threads to execute is not constant
  - Many threads available
    - System with many lightweight cores achieves a better throughput
  - Few threads available
    - System with aggressive cores achieves a better throughput
  - Single-thread performance is always better with aggressive cores

- Asymmetric Chip Multiprocessors (ACMPs):
  - Adapt to the variability in the number of threads
  - Limited in that there is no dynamic adaptation

- To provide dynamic adaptation:
  - We use core coupling



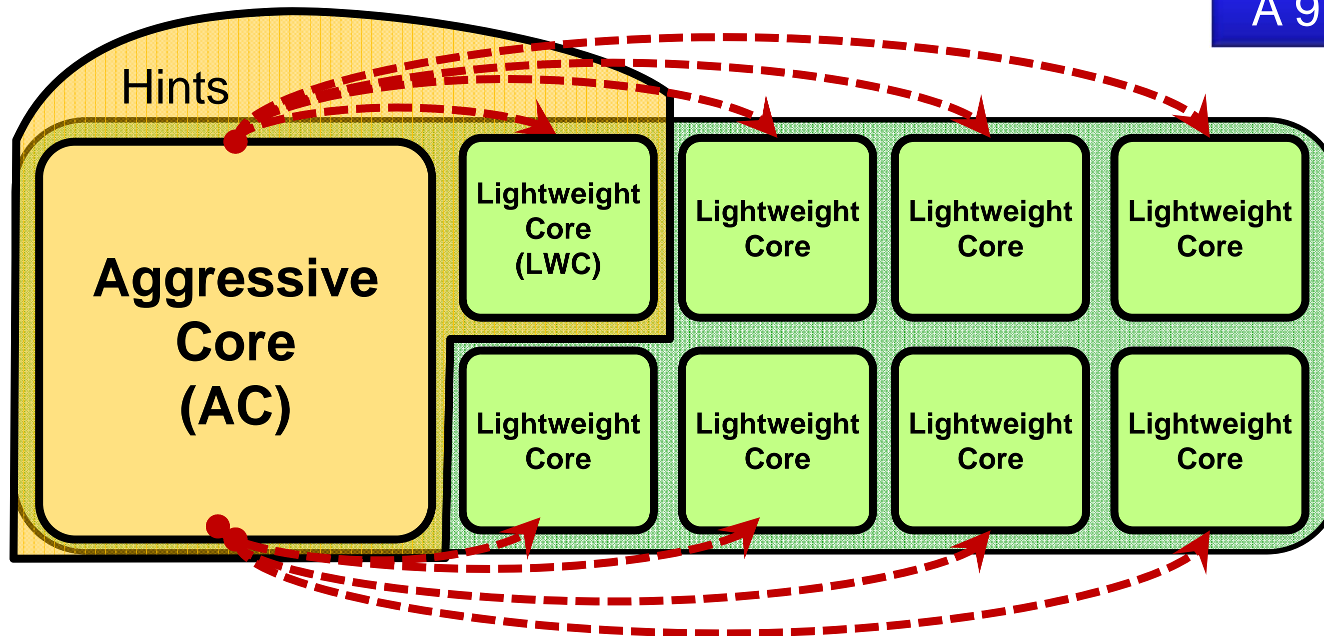
Core Coupling

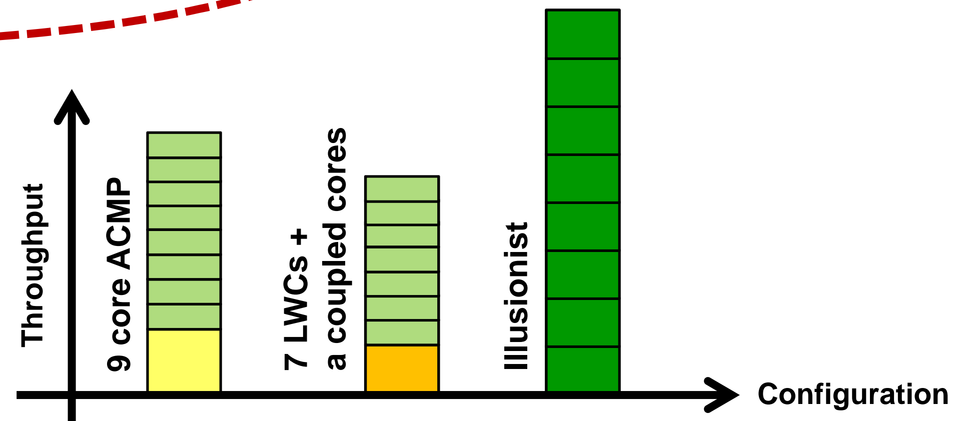
- Typically configured as leader/follower cores where the leader runs ahead and attempts to accelerates the follower

Techniques	Description
○ Slipstream ○ Master/slave Speculation	*The leader runs ahead by executing a "pruned" version of the application*
○ Flea Flicker ○ Dual-core Execution	*The leader speculates on long-latency operations*
○ Paceline	*The leader is aggressively frequency scaled (reduced safety margins)*
○ DIVA	*A smaller follower core simplifies the design/verification of the leader core*

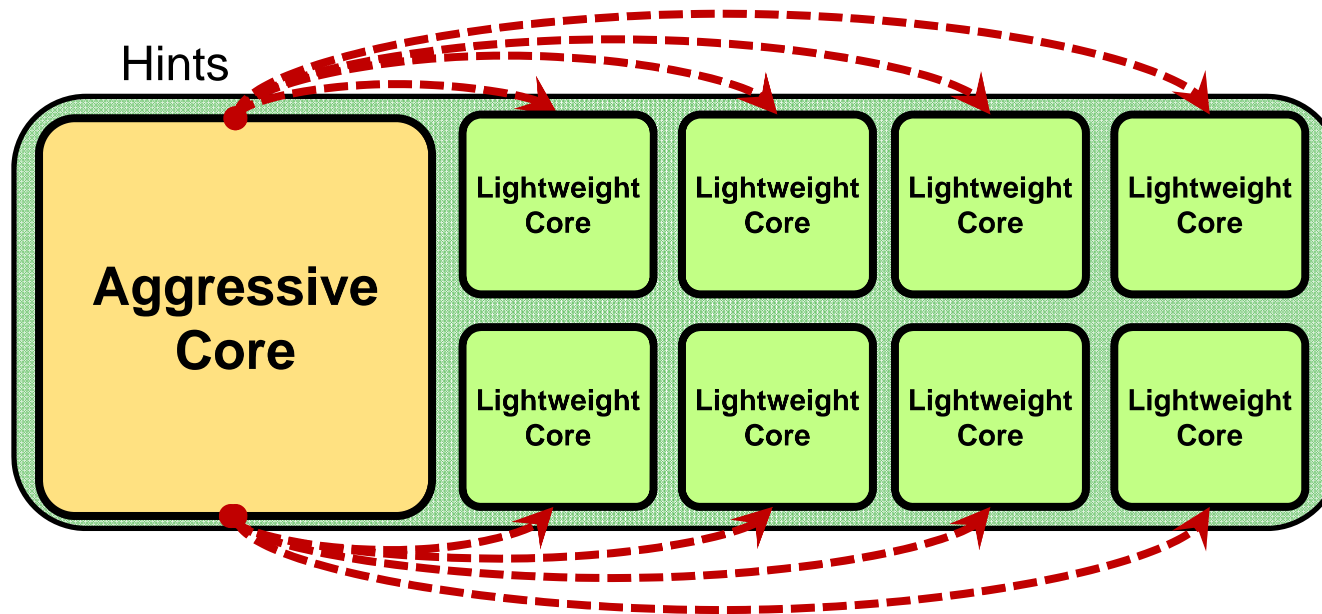
Extending Core Coupling

A 9 Core ACMP System



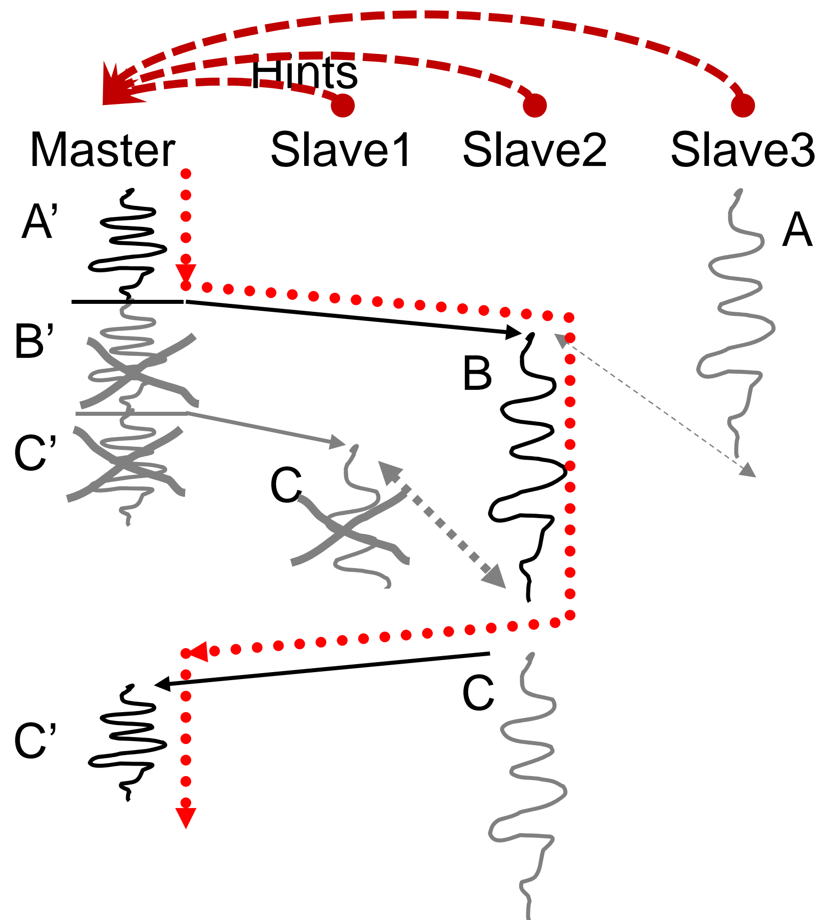


Illusionist vs Prior Work



- Higher single-thread performance for all LWCs
  - By using a single aggressive core
  - Giving the appearance of 8 semi-aggressive cores

Illusionist vs Prior Work



Master Slave Parallelization [Zilles'02]

- Higher single-thread performance for only a single aggressive core
- By using an army of LWCs (slave cores)
- Pushing the ILP limit
- Spawning threads for the slave cores to work on and also check the speculative computation on the master core

Providing Hints for Many Cores

- Original IPC of the aggressive core ~2X of that of a LWC
- We want an AC to keep up with a large number of LWCs
  - We need to substantially reduce the amount of work that the aggressive core needs to do per each thread running on a LWC
- We need to run lower num of instructions per each thread
  - We distill the program that the aggressive core needs to run
  - We limit the execution of the program only to most fruitful parts
- The main challenge here is to
  - Preserve the effectiveness of the hints while removing instructions

Program Distillation

- Objective: reduce the size of program while preserving the effectiveness of the original hints (branch prediction and cache hits)

- Distillation techniques
  - Aggressive instruction removal (on average, 77%)
    - Remove instructions which do not contribute to hint generation
    - Remove highly biased branches and their back slice
    - Remove memory inst. accessing the same cache line
  - Select the most promising program phases
    - Predictor that uses performance counters
    - Regression model based on IPC, $ and BP miss rates

Example of Instruction Removal

179.art

Original code

```
if (high<=low)
    return;

srand(10);
for (i=low;i<high;i++) {
  for (j=0;j<numf1s;j++) {
    if (i%low) {
      tds[j][i] = tds[j][0];
      tds[j][i] = bus[j][0];
    } else {
      tds[j][i] = tds[j][1];
      tds[j][i] = bus[j][1];
    }
  }
}

for (i=low;i<high;i++) {
  for (j=0;j<numf1s;j++) {
    noise1 = (double)(rand()&0xffff);
    noise2 = noise1/(double)0xffff;
    tds[j][i] += noise2;
    bus[j][i] += noise2;
  }
}
```

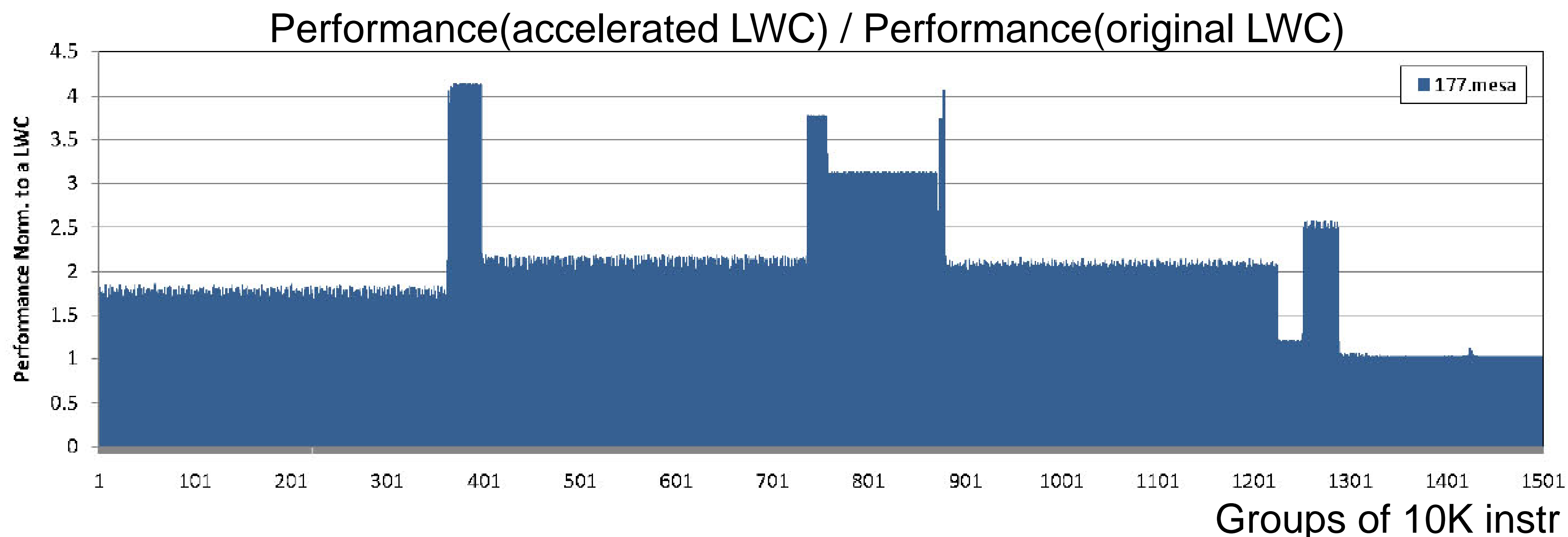
```
…
for (i=low;i<high;i=i+4) {
  for (j=0;j<numf1s;j++) {
    noise1 = (double)(rand()&0xffff);
    noise2 = noise1/(double)0xffff;
    tds[j][i] += noise2;
    bus[j][i] += noise2;
  }
  for (j=0;j<numf1s;j++) {
    noise1 = (double)(rand()&0xffff);
    noise2 = noise1/(double)0xffff;
    tds[j][i+1] += noise2;
    bus[j][i+1] += noise2;
  }
  for (j=0;j<numf1s;j++) {
    noise1 = (double)(rand()&0xffff);
    noise2 = noise1/(double)0xffff;
    tds[j][i+2] += noise2;
    bus[j][i+2] += noise2;
  }
  for (j=0;j<numf1s;j++) {
    noise1 = (double)(rand()&0xffff);
    noise2 = noise1/(double)0xffff;
    tds[j][i+3] += noise2;
    bus[j][i+3] += noise2;
  }
}
```

Distilled code

```
srand(10);
for (i=low;i<high;i=i+4) {
  for (j=0;j<numf1s;j++) {
    tds[j][i] = tds[j][1];
    tds[j][i] = bus[j][1];
  }
}

for (i=low;i<high;i=i+4) {
  for (j=0;j<numf1s;j++) {
    tds[j][i] = noise2;
    bus[j][i] = noise2;
  }
}
```

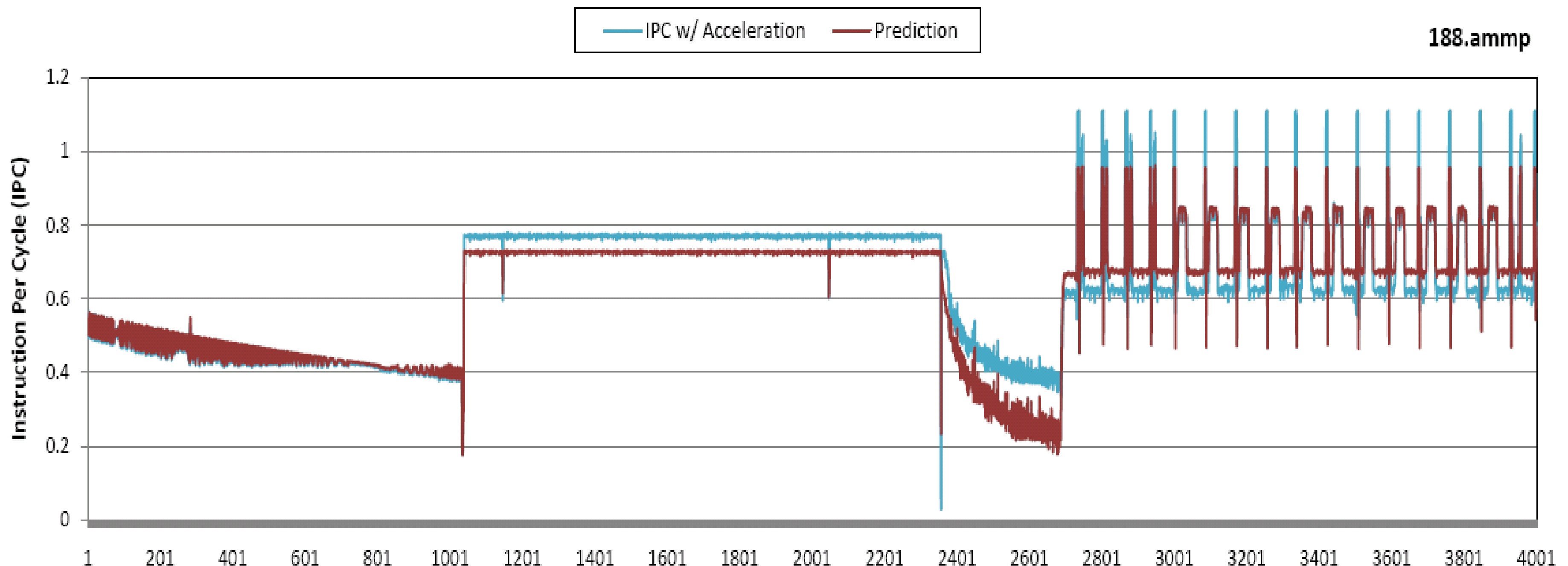
Hint Phases



If we can predict these phases without actually running the program on both lightweight and aggressive cores, we can limit the dual core execution only to the most useful phases

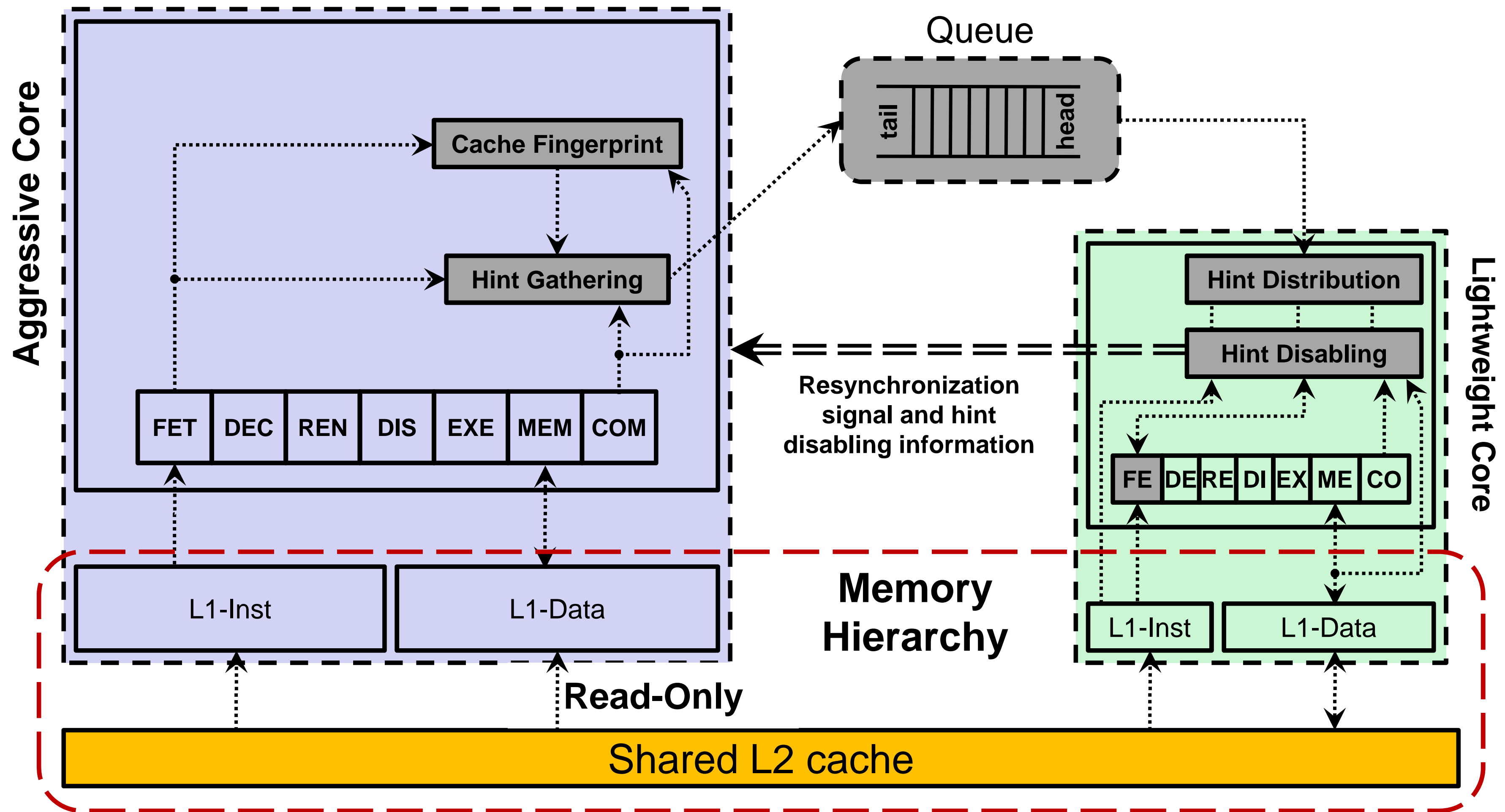
i-acoma group
ILLINOIS

Phase Prediction

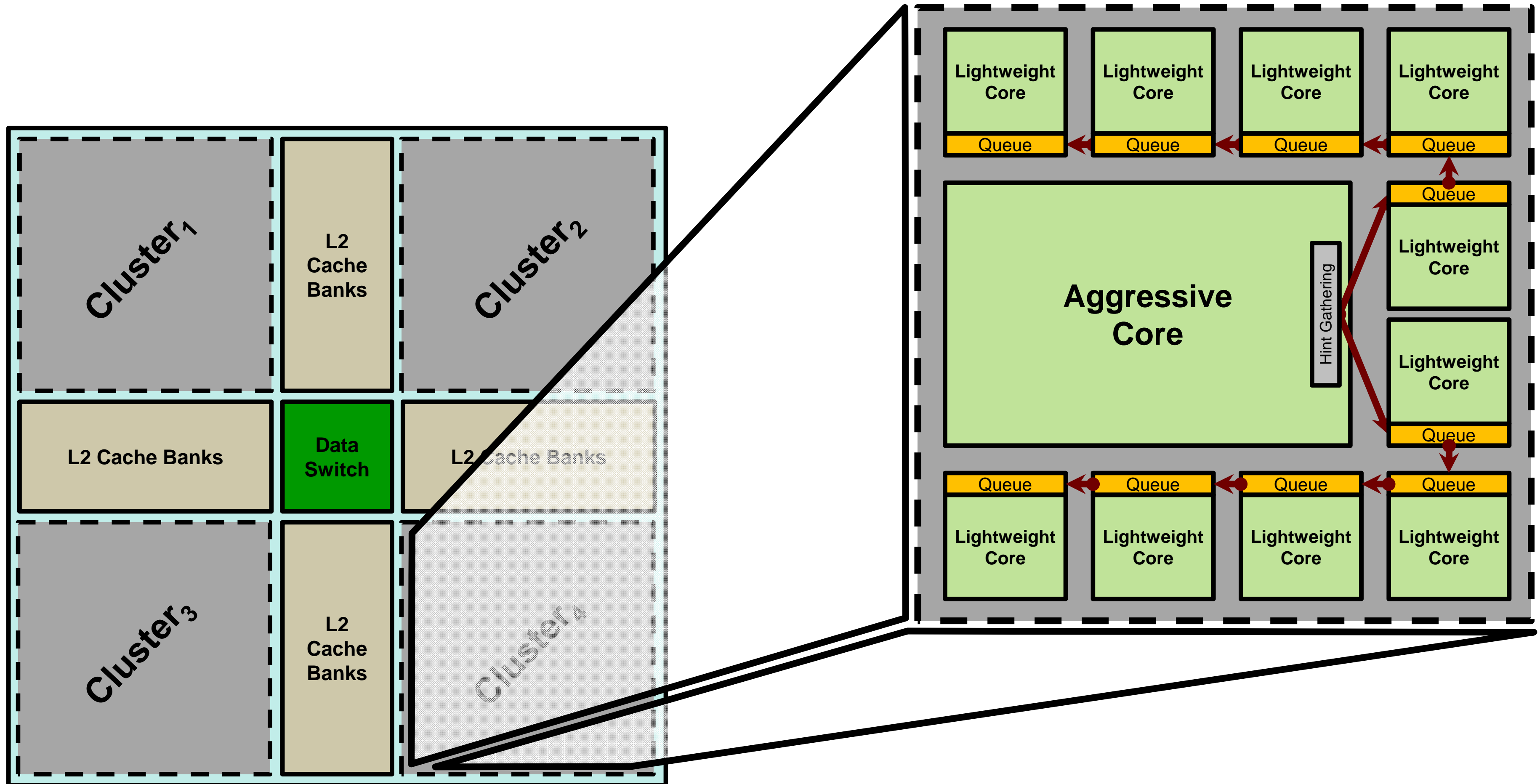


- Phase predictor :
  - does a decent job predicting the IPC trend
  - can sit either in the hypervisor or operating system and reads the performance counters while the threads running
- Aggressive core runs the thread that will benefit the most

Illusionist: Core Coupling Architecture



Illusionist System

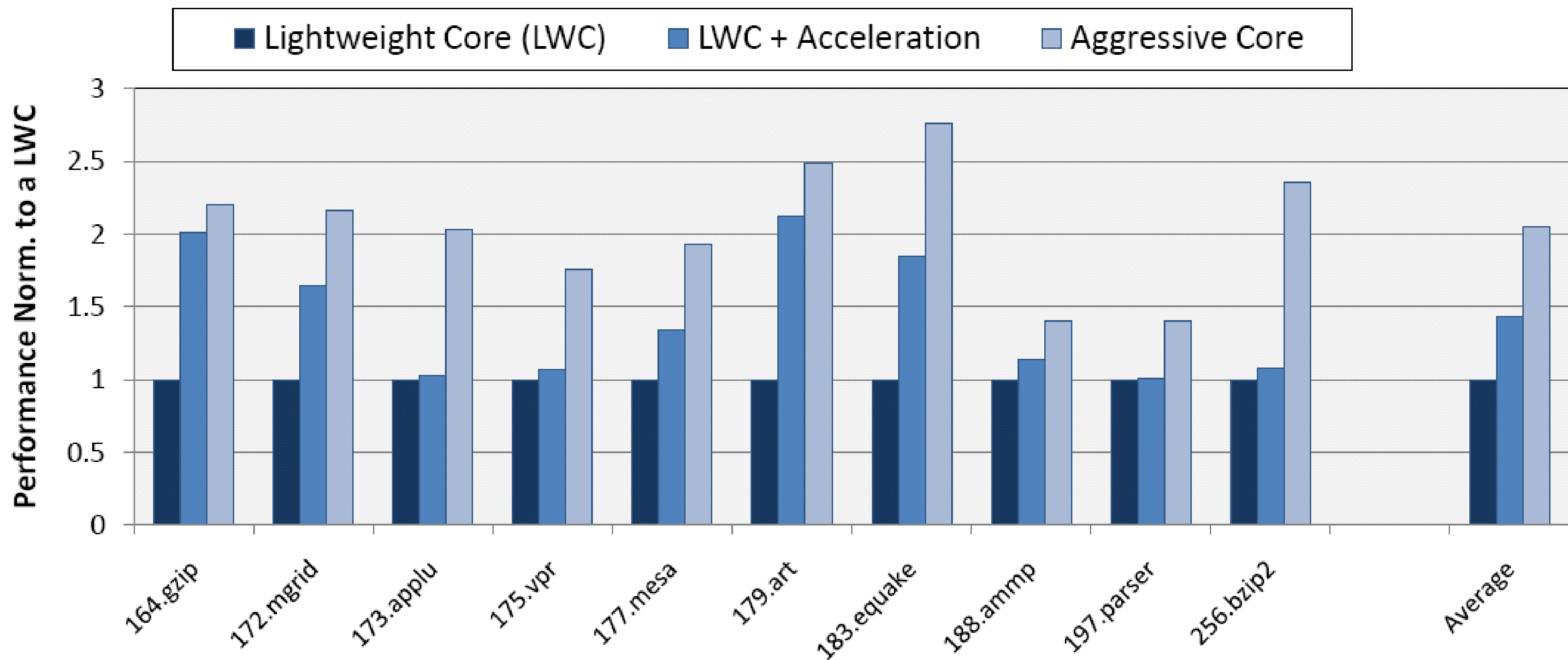


Experimental Methodology

- **Performance :** Heavily modified SimAlpha
  - Instruction removal and phase-based program pruning
  - SPEC-CPU-2K with SimPoint
- **Power :** Wattch, HotLeakage, and CACTI
- **Area :** Synopsys toolchain + 90nm TSMC

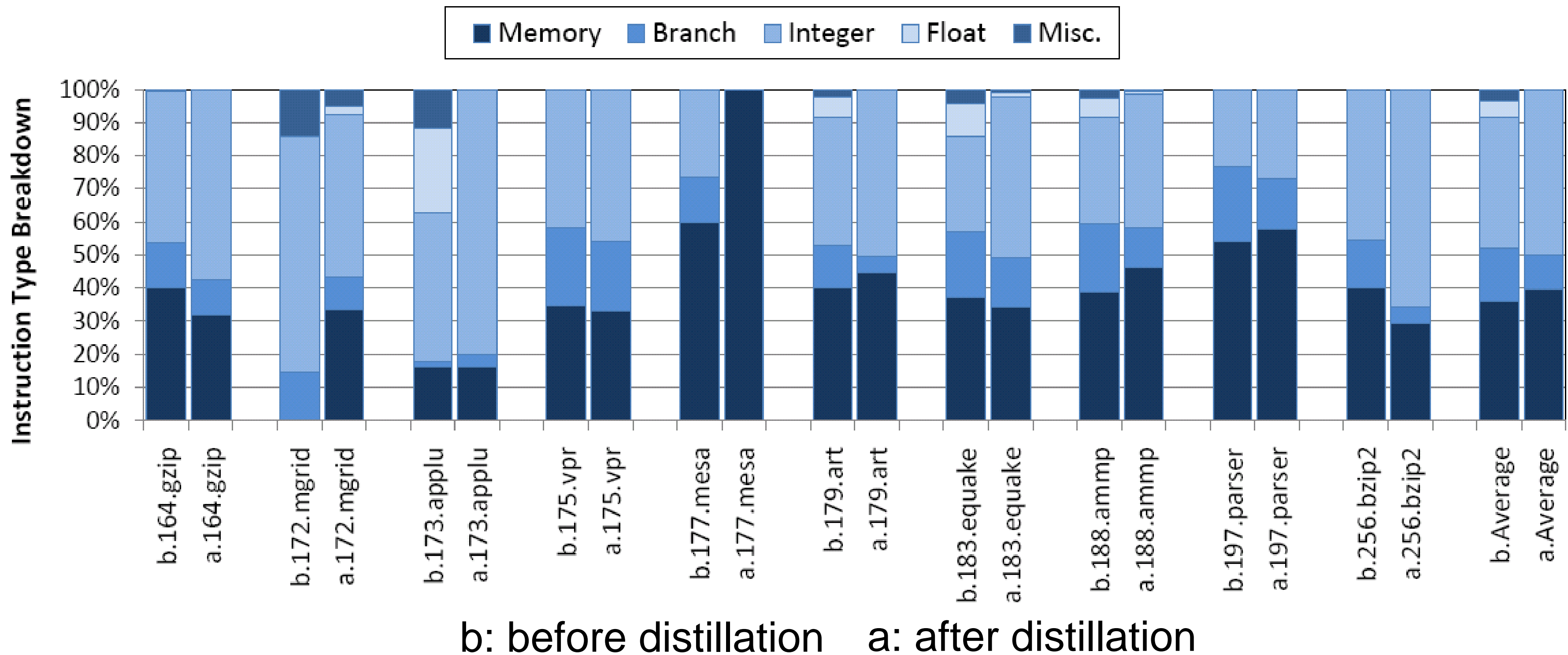
Parameter	A lightweight core	An aggressive core
Fetch/issue/commit width	2 per cycle	6 per cycle
Reorder buffer	32 entries	128 entries
Load/store queue entries	8/8	32/32
Issue queue	16 entries	64 entries
Instruction fetch queue	8 entries	32 entries
Branch predictor	tournament (bimodal + Illusionist BP)	tournament (bimodal + 2-level)
Branch target buffer size	256 entries, direct-map	1024 entries, 2-way
Branch history table	1024 entries	4096 entries
Return address stack	-	32 entries
L1 data cache	8KB direct-map, 3 cycles access latency, 2 ports	64KB, 4-way, 5 cycles access latency, 4 ports
L1 instr. cache	4KB direct-map, 2 cycles access latency, 2 ports	64KB, 4-way, 5 cycles access latency, 1 port
L2 cache	1MB per core, unified and shared, 8-way, 16 cycles access latency	
Main memory	250 cycles access latency	

Performance After Acceleration



On average, 43% speedup compared to a LWC

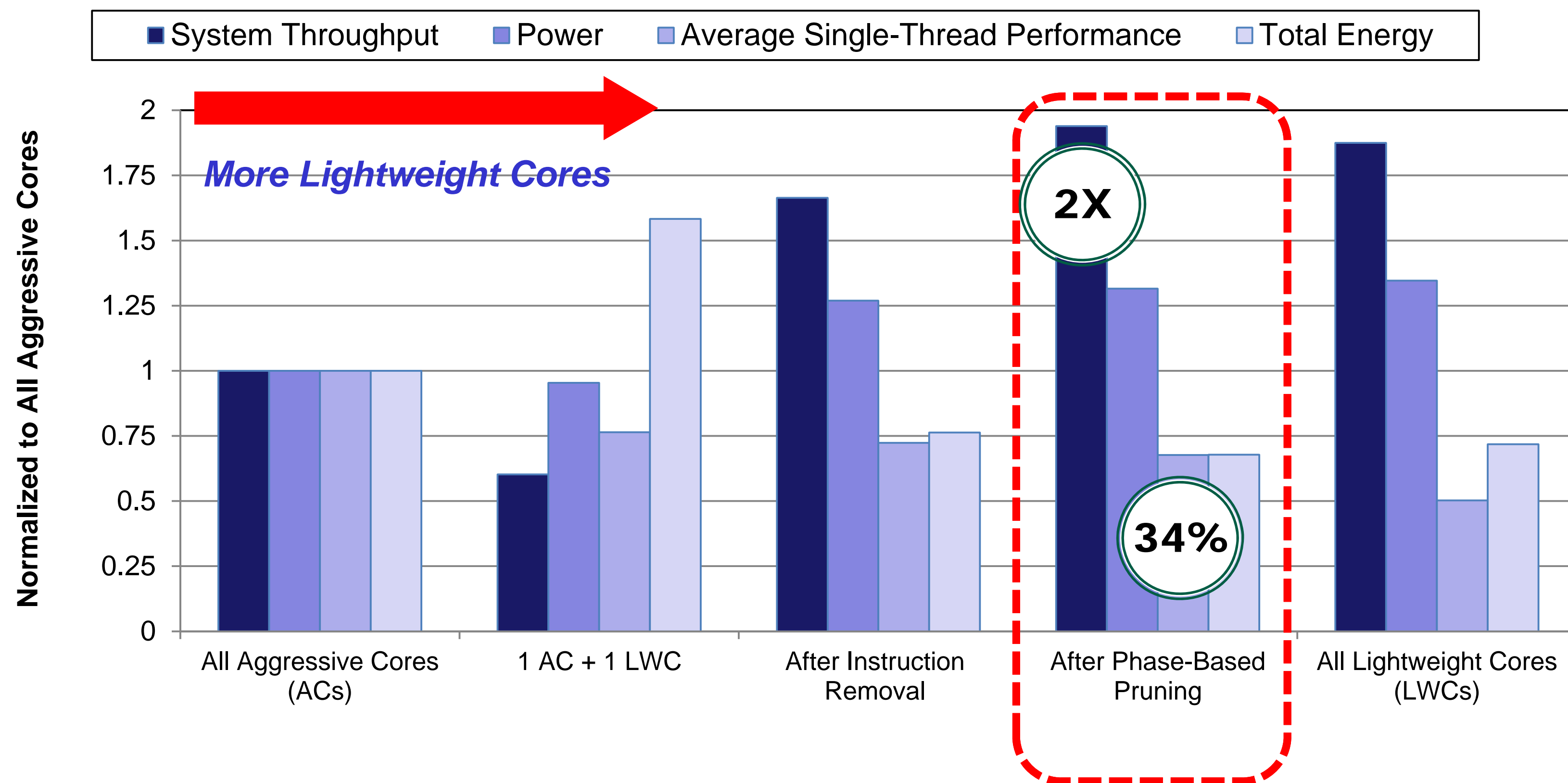
Instruction Type Breakdown



b: before distillation a: after distillation

In most benchmarks, the breakdowns are **similar**.

Area-Neutral Comparison of Alternatives



Conclusion

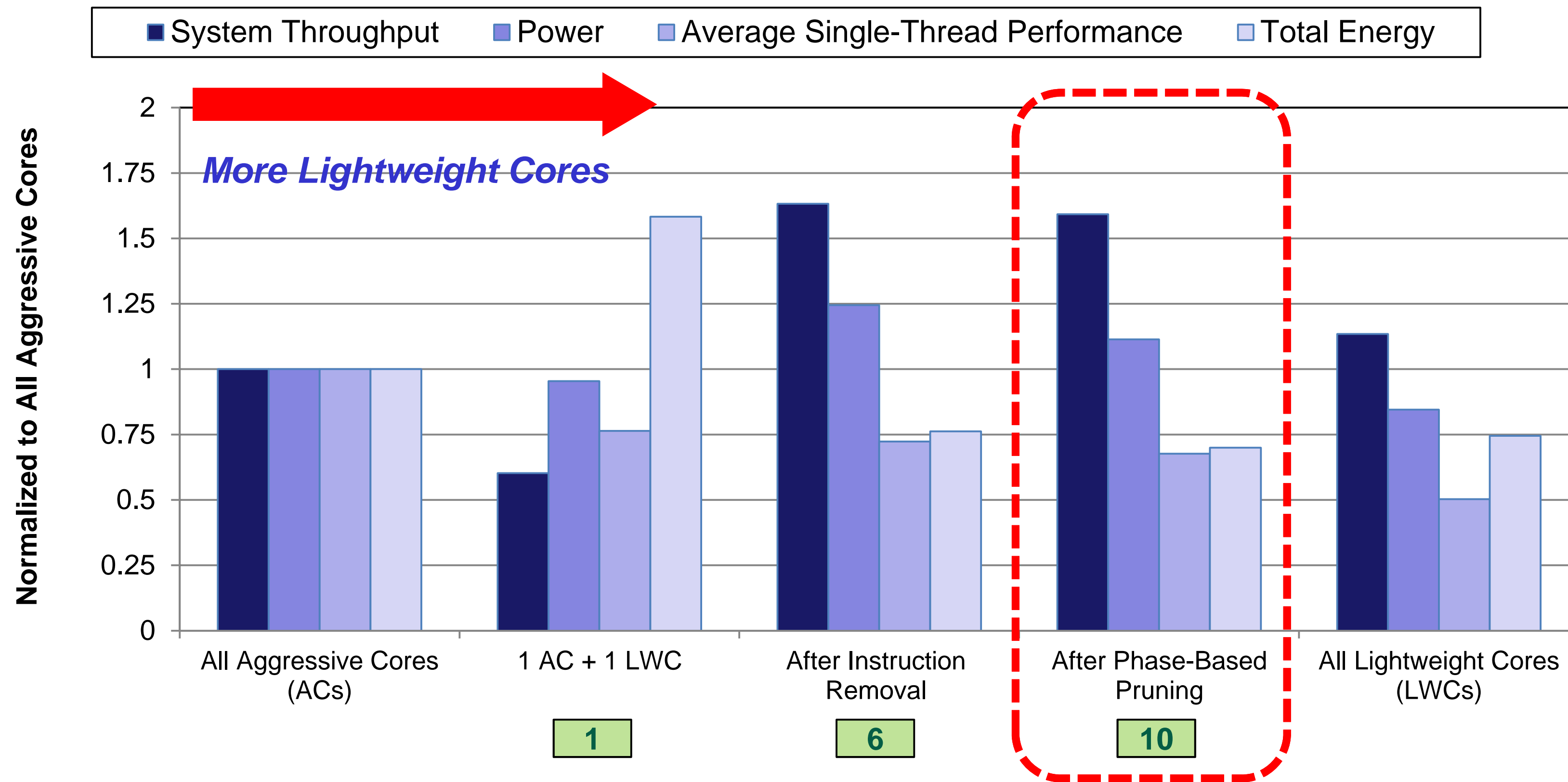
- On-demand acceleration of lightweight cores
  - using a few aggressive cores

- Aggressive core keeps up with many LWCs by
  - **Aggressive inst. removal** with a minimal impact on the hints
  - **Phase-based program pruning** based on hint effectiveness

- Illusionist provides an interesting design point
  - Compared to a CMP with only lightweight cores
    - 35% better single thread performance per thread
  - Compared to a CMP with only aggressive cores
    - 2X better system throughput

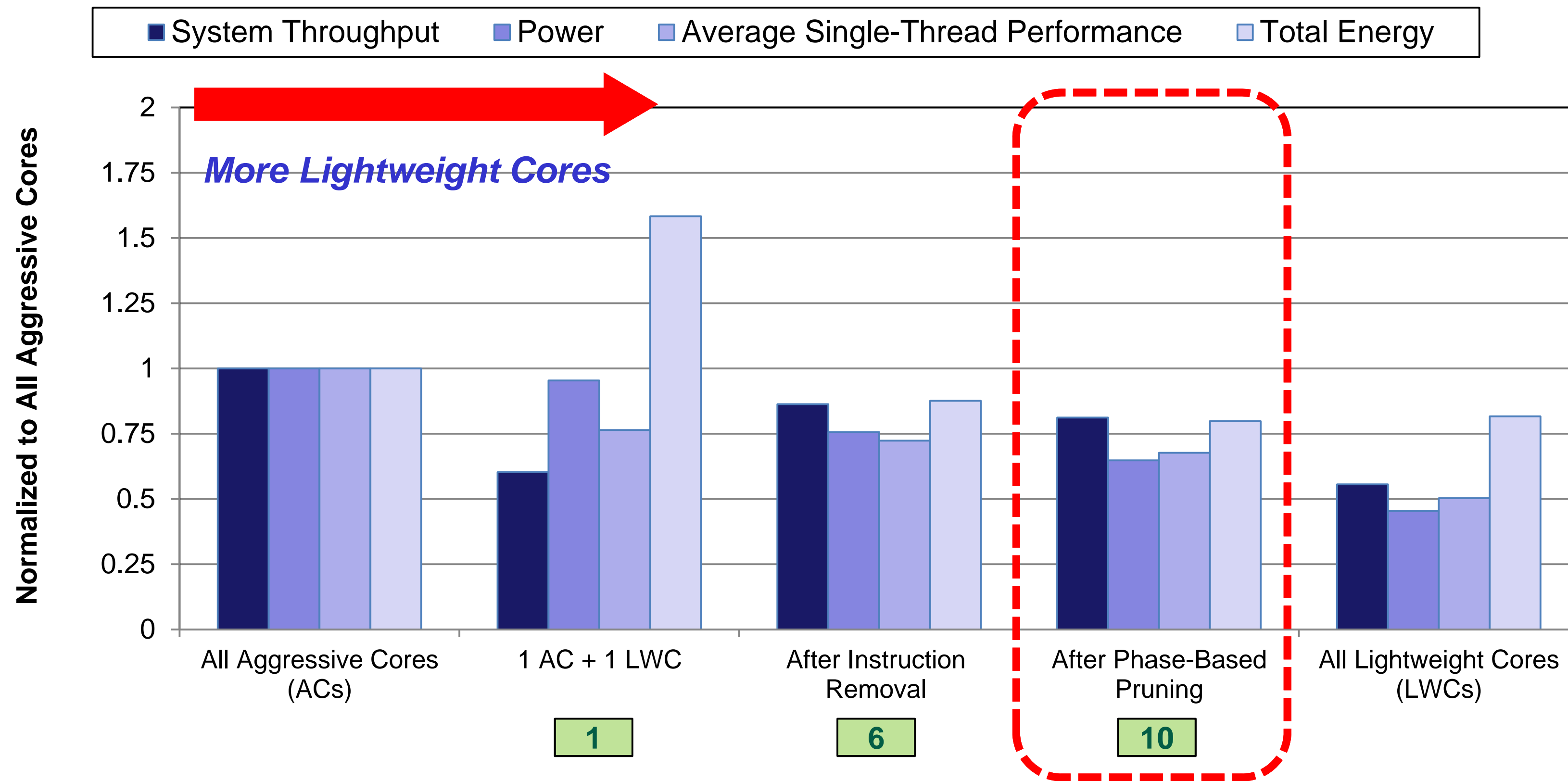
BACKUP SLIDES

Comparison with Alternatives



number of available threads = 60% of the number of lightweight cores

Comparison with Alternatives

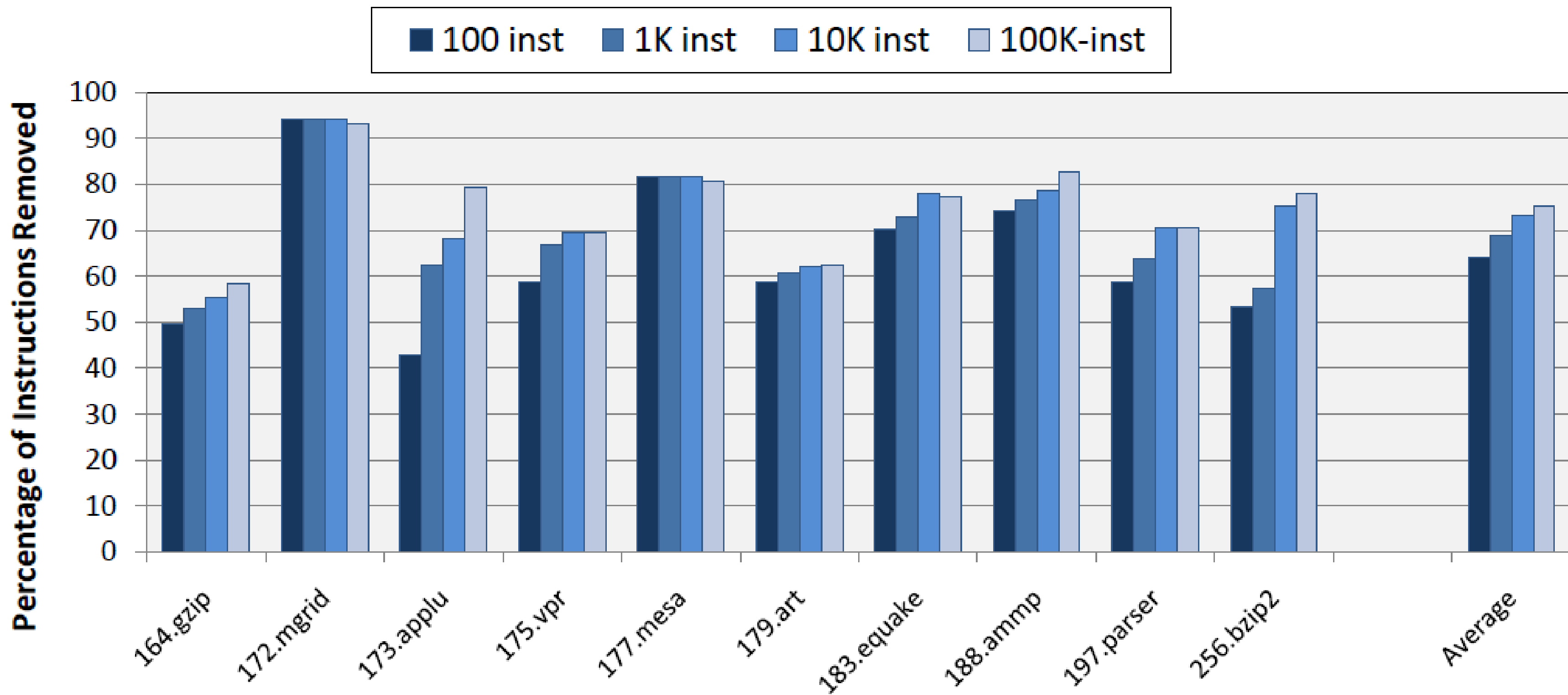


number of available threads = 30% of the number of lightweight cores

i-acoma group

ILLINOIS

Percentage of Instruction Removed



Hint Accuracy after Instruction Removal