

Kismet: Parallel Speedup Estimates for Serial Programs

Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor

*Computer Science and Engineering
University of California, San Diego*



Questions in Parallel Software Engineering



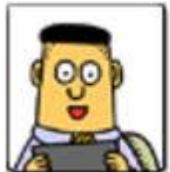
I heard about these new-fangled multicore chips.
How much faster will PowerPoint be with 128 cores?



We wasted 3 months for 0.5% parallel speedup.
Can't we get parallel performance estimates earlier?



How can I set the parallel performance goals
for my intern, Asok?



Dilbert asked me to achieve 128X speedup.
How can I convince him it is impossible
without changing the algorithm?

Kismet Helps Answer These Questions

Kismet automatically provides the ***estimated parallel speedup upperbound*** from ***serial source code***.

```
$> make CC=kismet-cc
```

1. Produce instrumented binary
with `kismet-cc`

```
$> $(PROGRAM) $(INPUT)
```

2. Perform parallelism profiling
with a sample input

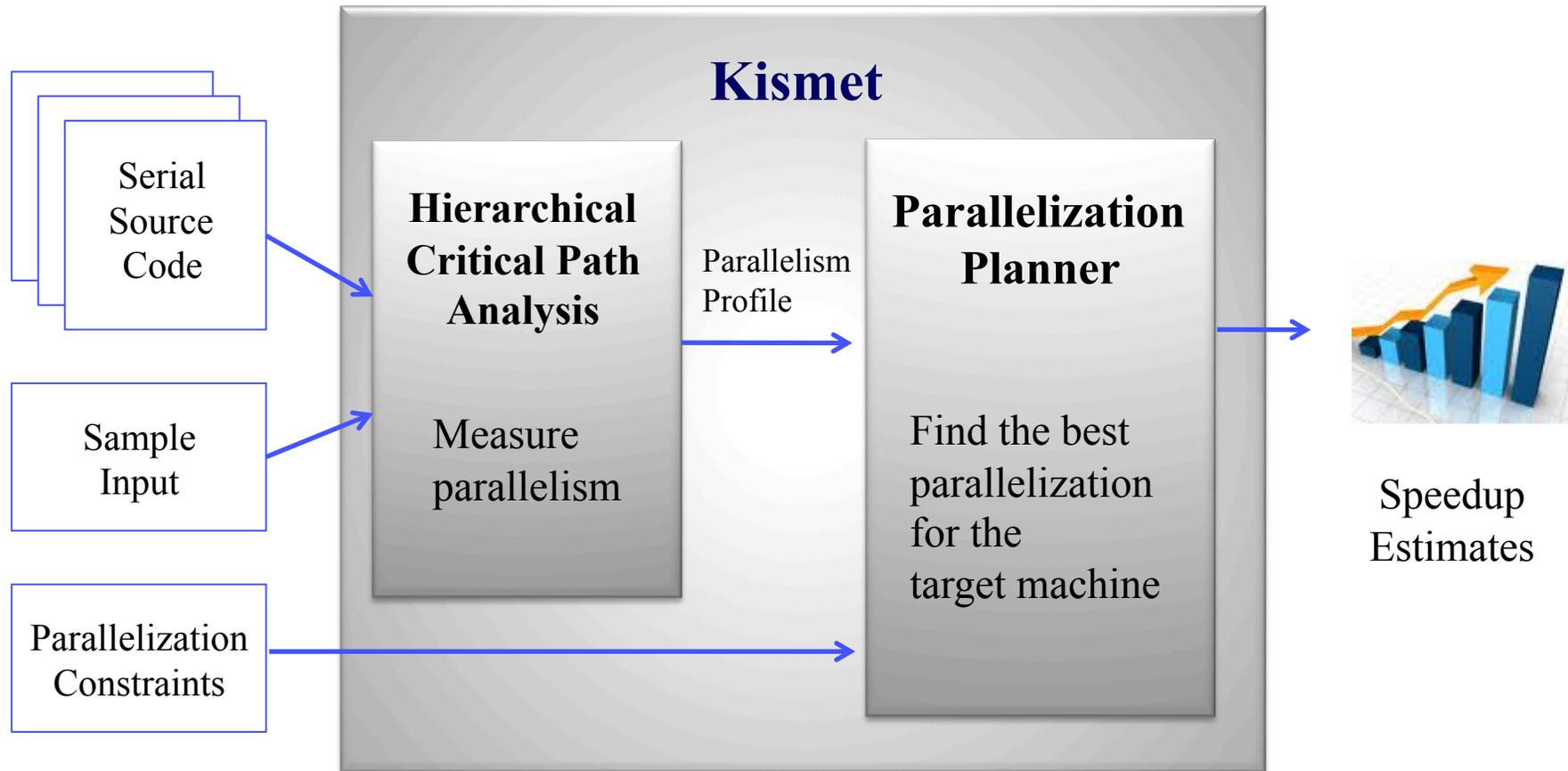
```
$> kismet -opteron -openmp
```

```
Cores    1    2    4    8    16    32  
Speedup  1    2    3.8  3.8  3.8  3.8  
(est.)
```

3. Estimate speedup
under given constraints

Kismet's easy-to-use usage model

Kismet Overview



Kismet extends critical path analysis to incorporate the constraints that affect real-world speedup.

Outline

- Introduction
- **Background: Critical Path Analysis**
- How Kismet Works
- Experimental Results
- Conclusion

The Promise of Critical Path Analysis (CPA)

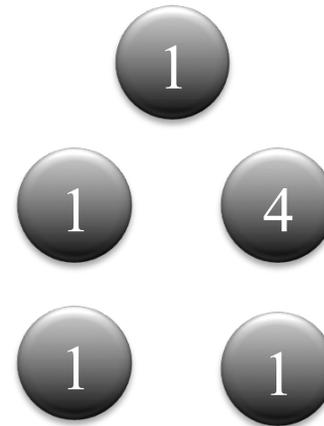
- **Definition:** program analysis that computes the longest dependence chain in the dynamic execution of a serial program
- **Typical Use:** approximate the upperbound on parallel speedup without parallelizing the code
- **Assumes:** an ideal execution environment
 - All parallelism exploitable
 - Unlimited cores
 - Zero parallelization overhead

How Does CPA Compute Critical Path?

```
la    $2, $ADDR
load  $3, $2(0)
addi  $4, $2, #4
store $4, $2(4)
store $3, $2(8)
```

How Does CPA Compute Critical Path?

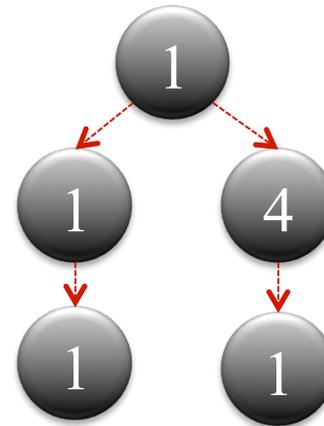
```
la    $2, $ADDR  
load  $3, $2(0)  
addi  $4, $2, #4  
store $4, $2(4)  
store $3, $2(8)
```



node: dynamic instruction with latency

How Does CPA Compute Critical Path?

```
la    $2, $ADDR
load  $3, $2(0)
addi  $4, $2, #4
store $4, $2(4)
store $3, $2(8)
```

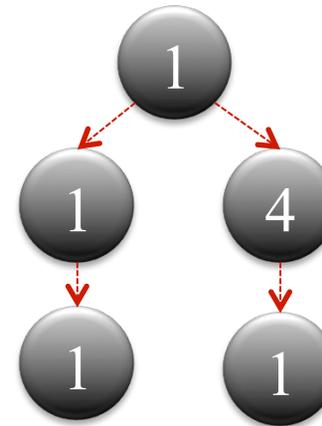


node: dynamic instruction with latency

edge: dependence between instructions

How Does CPA Compute Critical Path?

```
la    $2, $ADDR
load  $3, $2(0)
addi  $4, $2, #4
store $4, $2(4)
store $3, $2(8)
```



work = 8

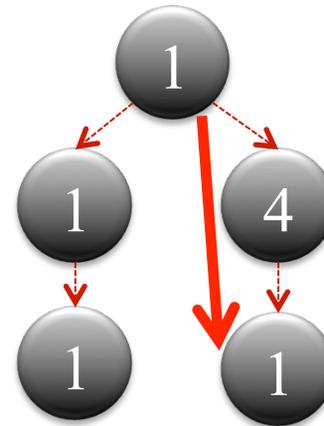
node: dynamic instruction with latency

edge: dependence between instructions

work: serial execution time,
total sum of node weights

How Does CPA Compute Critical Path?

```
la    $2, $ADDR
load  $3, $2(0)
addi  $4, $2, #4
store $4, $2(4)
store $3, $2(8)
```



work = 8

↓ cp = 6

node: dynamic instruction with latency

edge: dependence between instructions

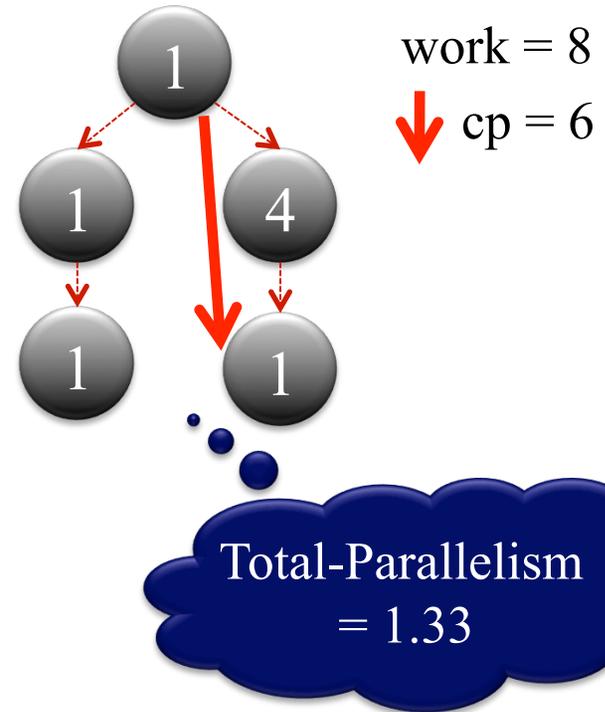
work: serial execution time,
total sum of node weights

critical path length (cp):
minimum parallel execution time

How Does CPA Compute Critical Path?

```

la    $2, $ADDR
load  $3, $2(0)
addi  $4, $2, #4
store $4, $2(4)
store $3, $2(8)
    
```



node: dynamic instruction with latency

edge: dependence between instructions

work: serial execution time,
 total sum of node weights

critical path length (cp):
 minimum parallel execution time

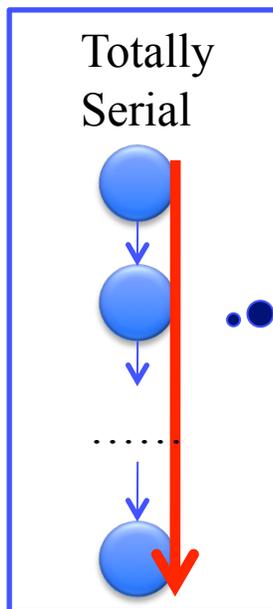
$$\text{Total-Parallelism} = \frac{\text{work}}{\text{critical path length}}$$

Total-Parallelism Metric: Captures the Ideal Speedup of a Program

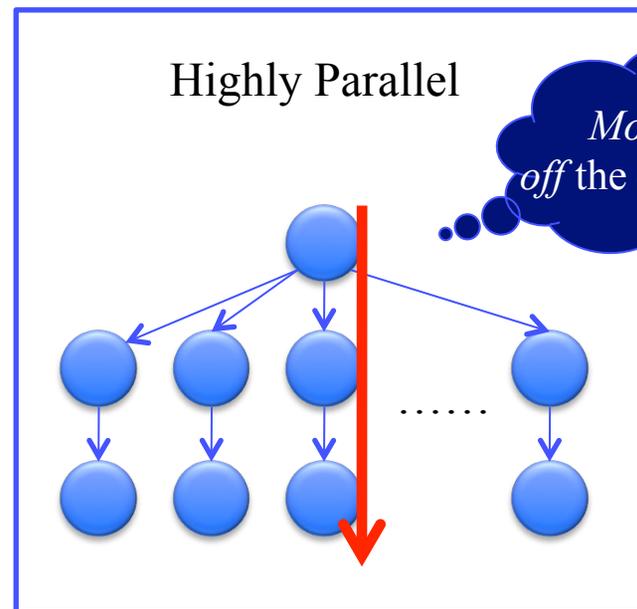


Min 1.0

Total-Parallelism



*All the work
on the critical path*



*Most work
off the critical path*

Why CPA is a Good Thing

- Works on original, unmodified serial programs
- Provides an approximate upperbound in speedup, after applying typical parallelization transformations
 - e.g. loop interchange, loop fusion, index-set splitting, ...
- Output is invariant of serial expression of program
 - Reordering of two independent statements does not change parallelism

A Brief History of CPA

- Employed to Characterize Parallelism in Research
 - COMET [Kumar '88]: Fortran statement level
 - Paragraph [Austin '92]: Instruction level
 - Limit studies for ILP-exploiting processors [Wall, Lam '92]
- Not widely used in programmer-facing parallelization tools

Why isn't CPA commonly used in programmer-facing tools?

Benchmark	Measured Speedup (16 cores)	CPA Estimated Speedup	Optimism Ratio
ep	15.0	9722	648
life	12.6	116278	9228
is	4.4	1300216	295503
sp	4.0	189928	47482
unstruct	3.1	3447	1112
sha	2.1	4.8	2.3

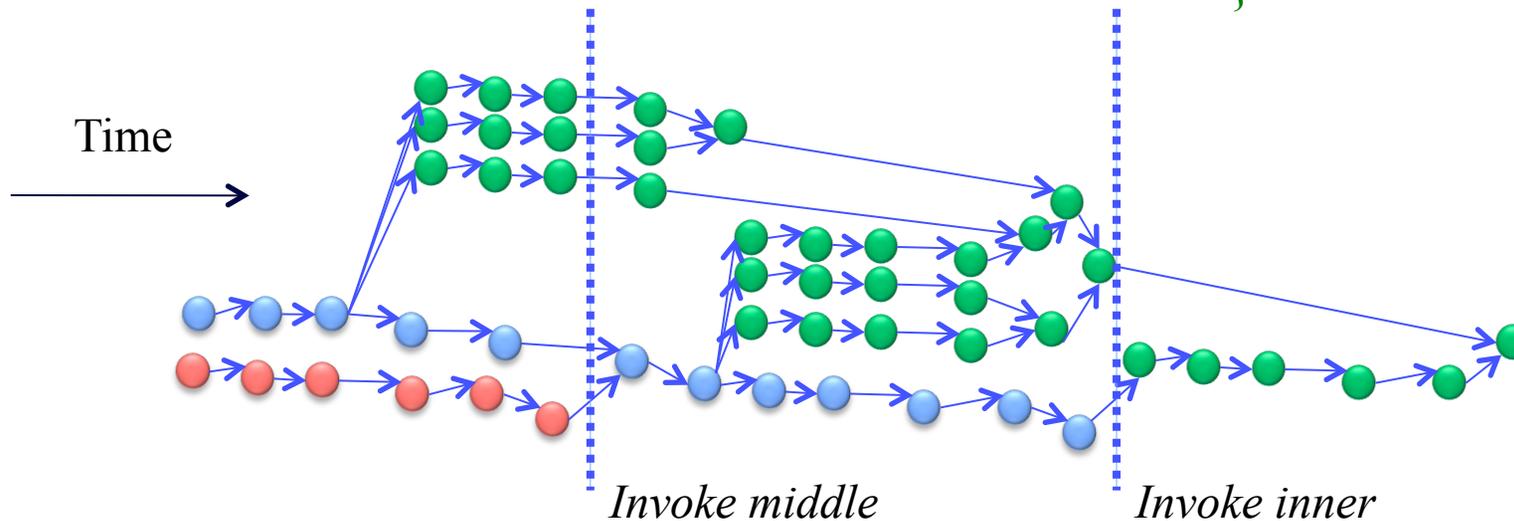
CPA estimated speedups do not correlate with real-world speedups.

CPA Problem #1: Data-flow Style Execution Model Is Unrealistic

```
void outer( )  
{  
  ....  
  middle();  
}
```

```
void middle( )  
{  
  ....  
  inner();  
}
```

```
void inner( )  
{  
  ....  
  parallel doall for-loop  
  reduction  
}
```



*Difficult to map this onto
von Neumann machine and imperative programming language*

CPA Problem #2:

Key Parallelization Constraints Are Ignored

Exploitability

What type of parallelism is supported by the target platform?
e.g. Thread Level (TLP), Data Level (DLP), Instruction Level (ILP)

Resource
Constraints

How many cores are available for parallelization?

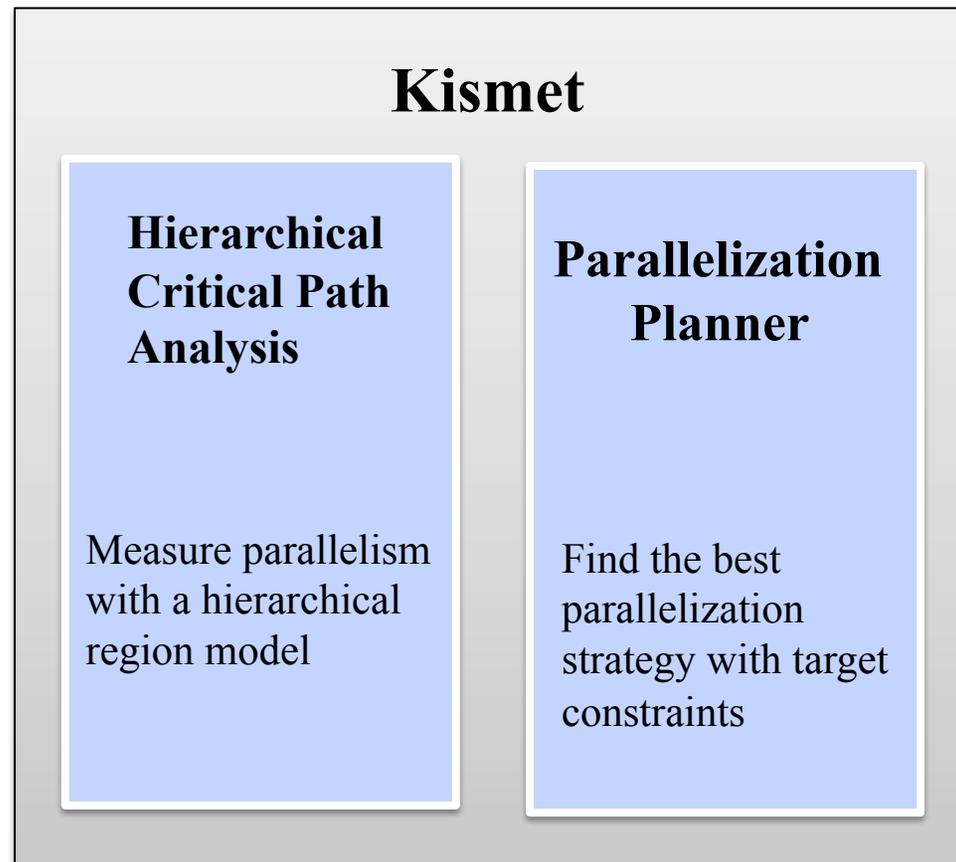
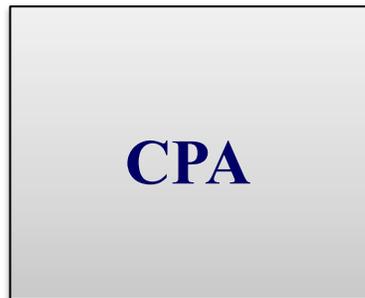
Overhead

Do overheads eclipse the benefit of the parallelism?
e.g. scheduling, communication, synchronization

Outline

- Introduction
- Background: Critical Path Analysis
- **How Kismet Works**
- Experimental Results
- Conclusion

Kismet Extends CPA to Provide Practical Speedup Estimates

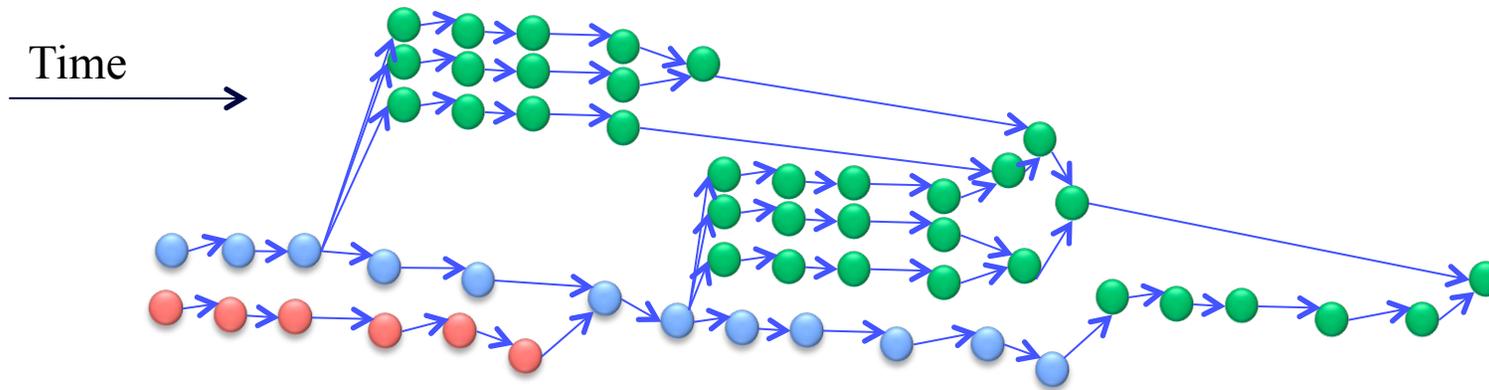


Revisiting CPA Problem #1: Data-flow Style Execution Model Is Unrealistic

```
void top()  
{  
  ....  
  middle();  
}
```

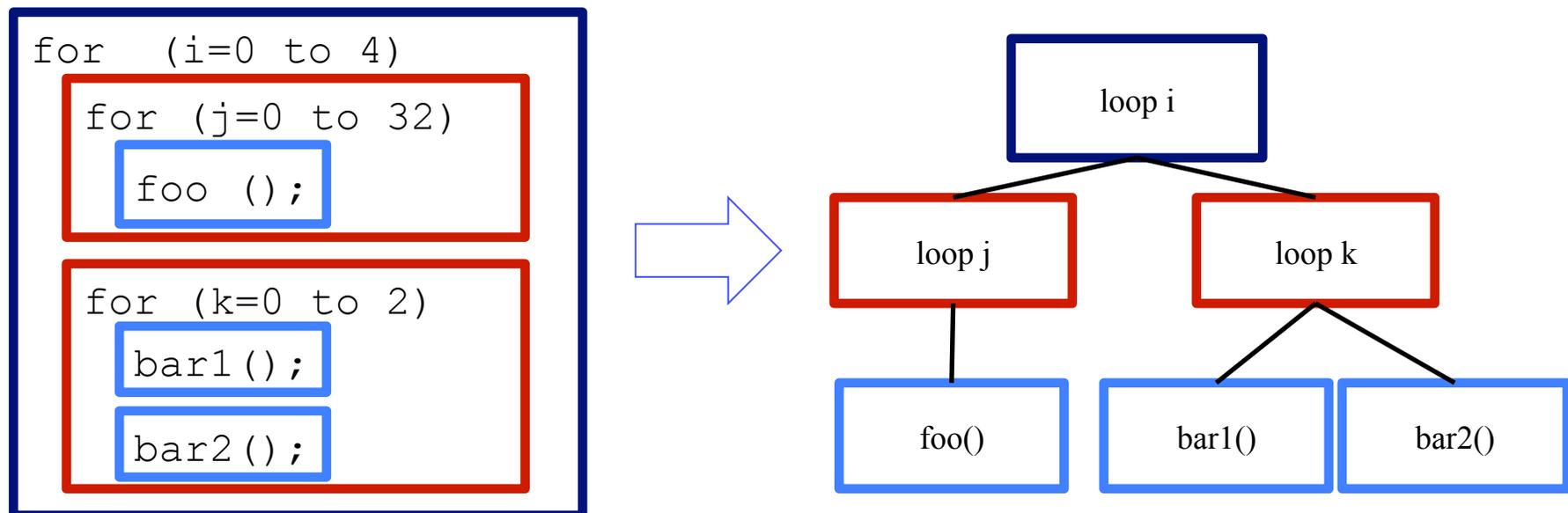
```
void middle()  
{  
  ....  
  inner();  
}
```

```
void inner()  
{  
  ....  
  parallel doall for-loop  
  reduction  
}
```



Hierarchical Critical Path Analysis (HCPA)

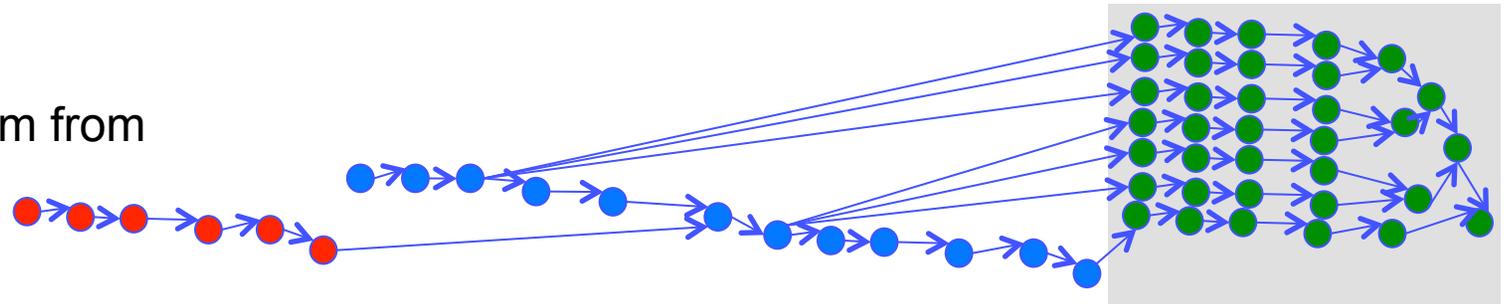
- Step 1. Model a program execution with hierarchical regions
- Step 2. Recursively apply CPA to each nested region
- Step 3. Quantify self-parallelism



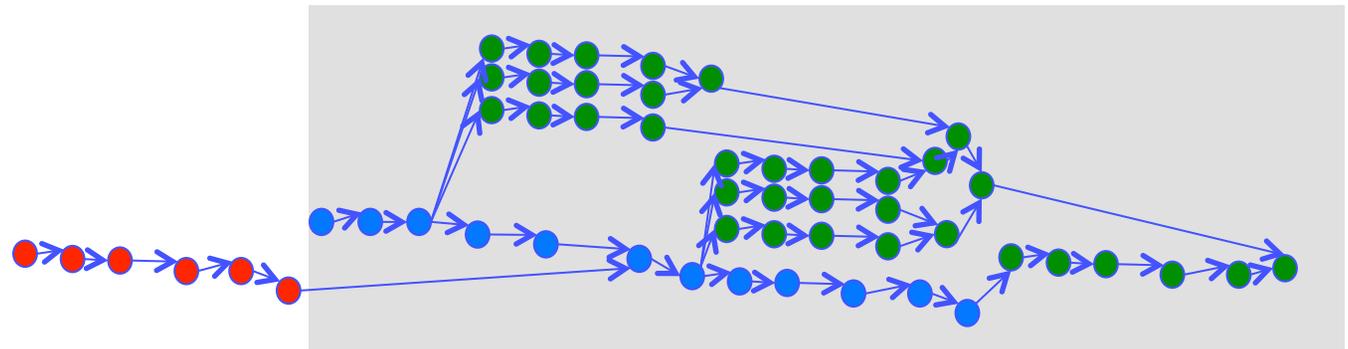
HCPA Step 1. Hierarchical Region Modeling

HCPA Step 2: Recursively Apply CPA

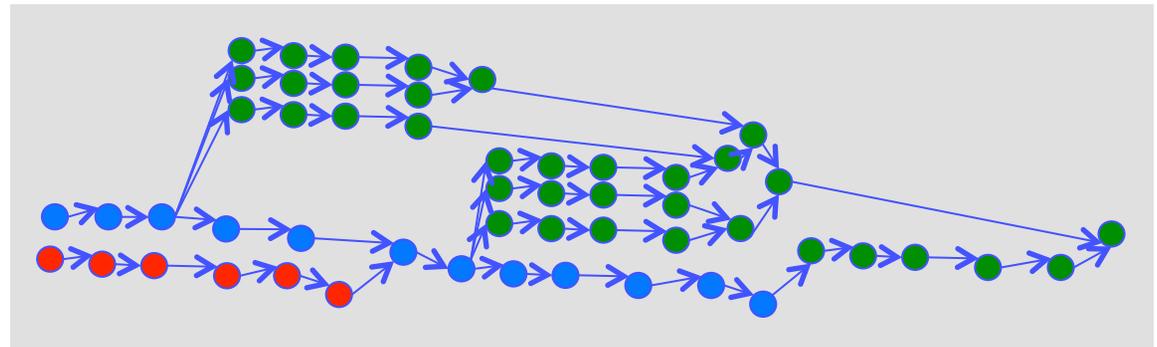
Total-Parallelism from
inner()
= $\sim 7X$



Total-Parallelism from
middle() and *inner()*
= $\sim 6X$



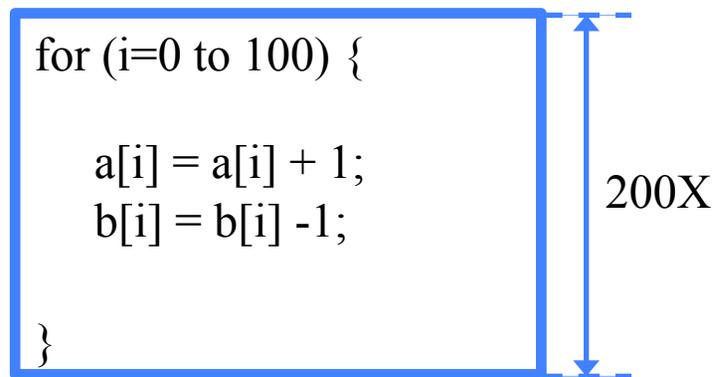
Total-Parallelism from
outer(), *middle()*, and *inner()*
= $\sim 5X$



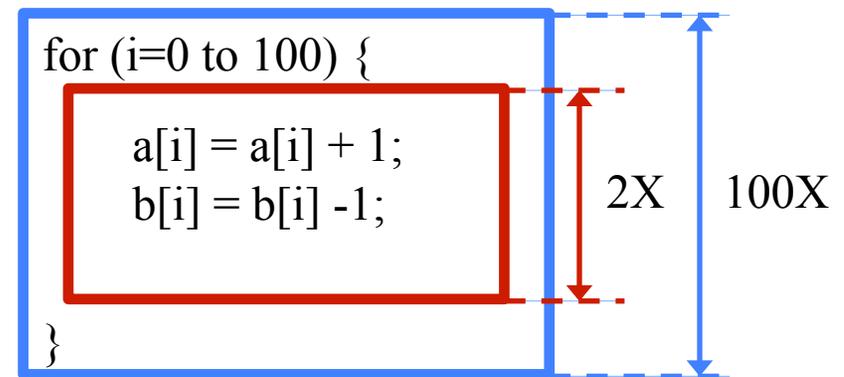
*What is a region's parallelism
excluding the parallelism from its nested regions?*

HCPA: Introducing *Self-Parallelism*

- Represents a region's ideal speedup
- Differentiates a parent's parallelism from its children's
- Analogous to *self-time* in serial profilers

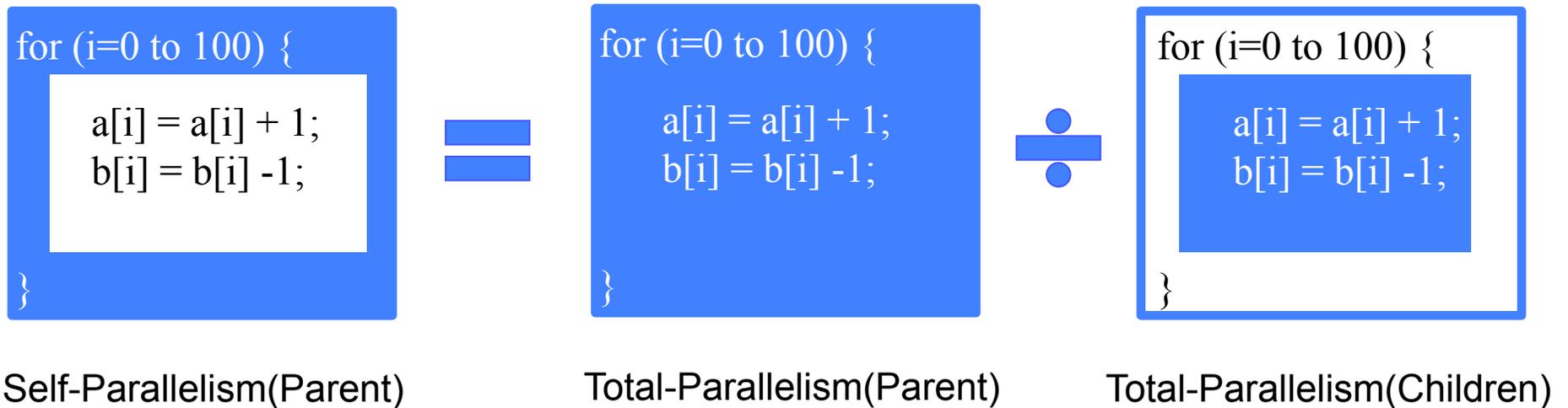


Total-Parallelism
(from CPA)



Self-Parallelism
(from HCPA)

HCPA Step 3: Quantifying Self-Parallelism

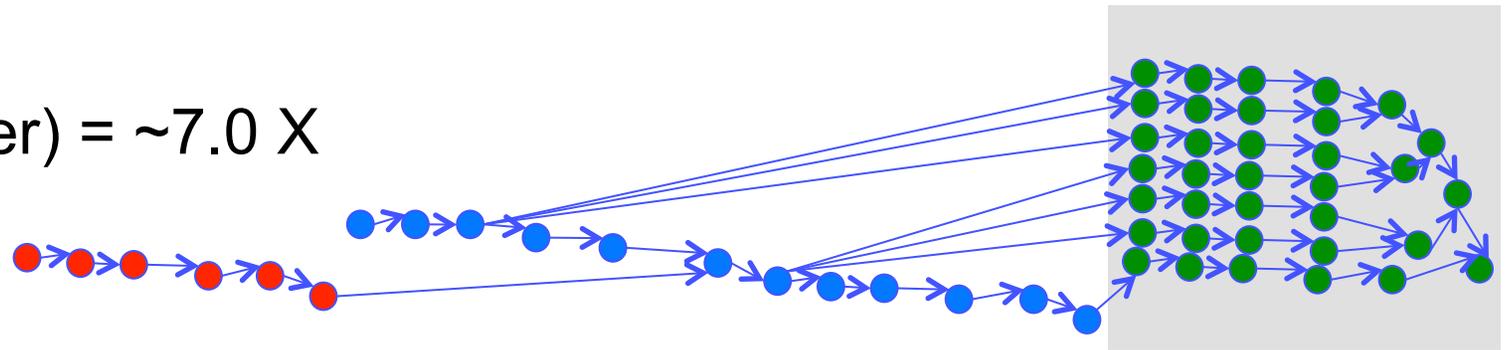


$$SP(R) = \begin{cases} \frac{\sum_{k=1}^n cp(child(R, k))}{cp(R)} & \text{R is a non-leaf} \\ \frac{work(R)}{cp(R)} & \text{R is a leaf} \end{cases}$$

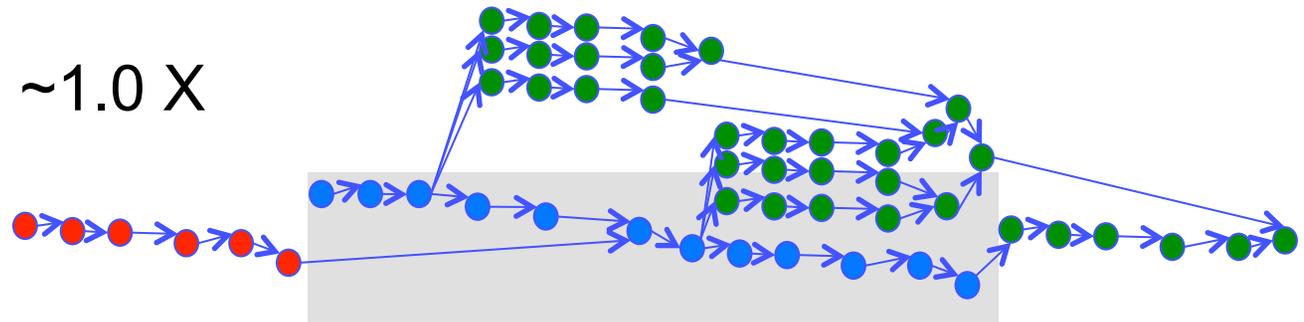
Generalized Self-Parallelism Equation

Self-Parallelism: Localizing Parallelism to a Region

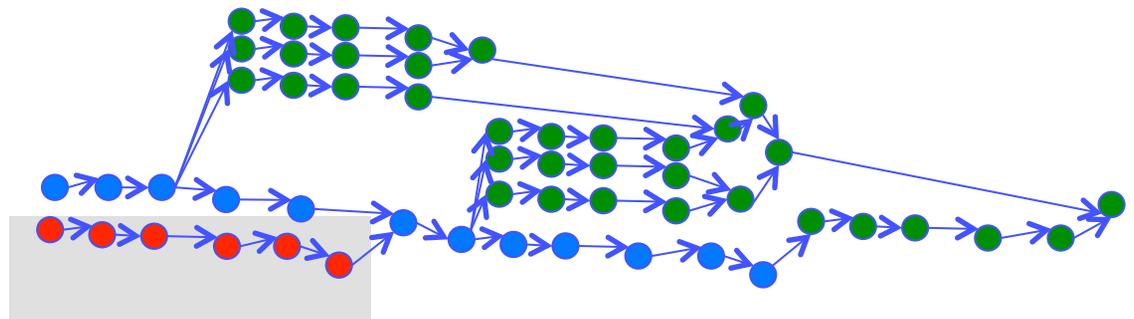
Self-P (inner) = $\sim 7.0 X$



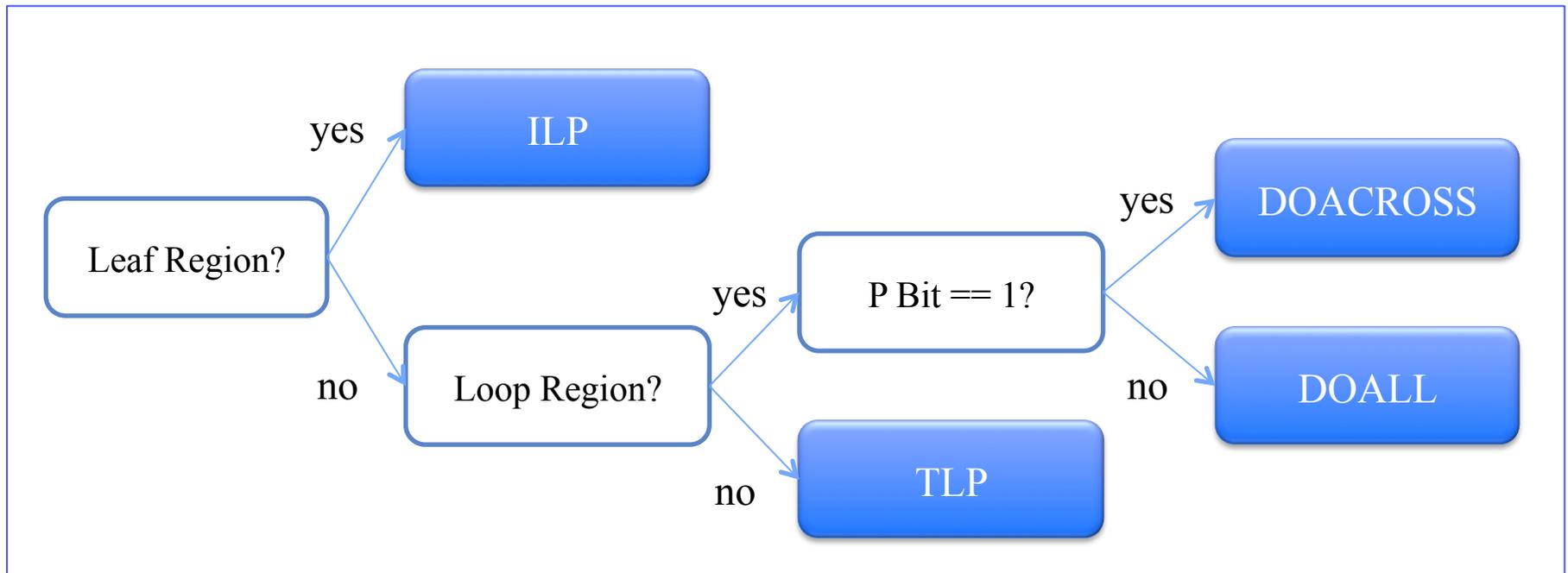
Self-P (middle) = $\sim 1.0 X$



Self-P (outer) = $\sim 1.0 X$



Classifying Parallelism Type



See our paper for details...

Why HCPA is an Even Better Thing

■ HCPA:

- Keeps all the desirable properties of CPA
- Localizes parallelism to a region via the self-parallelism metric and hierarchical region modeling
- Facilitates the classification of parallelism
- Enables more realistic modeling of parallel execution (see next slides)

Outline

- Introduction
- Background: Critical Path Analysis
- **How Kismet Works**
 - HCPA
 - **Parallelization Planner**
- Experimental Results
- Conclusion

Revisiting CPA Problem #2: Key Parallelization Constraints Are Ignored

Exploitability

What type of parallelism is supported by the target platform?
e.g. Thread Level (TLP), Data Level (DLP), Instruction Level (ILP)

Resource
Constraints

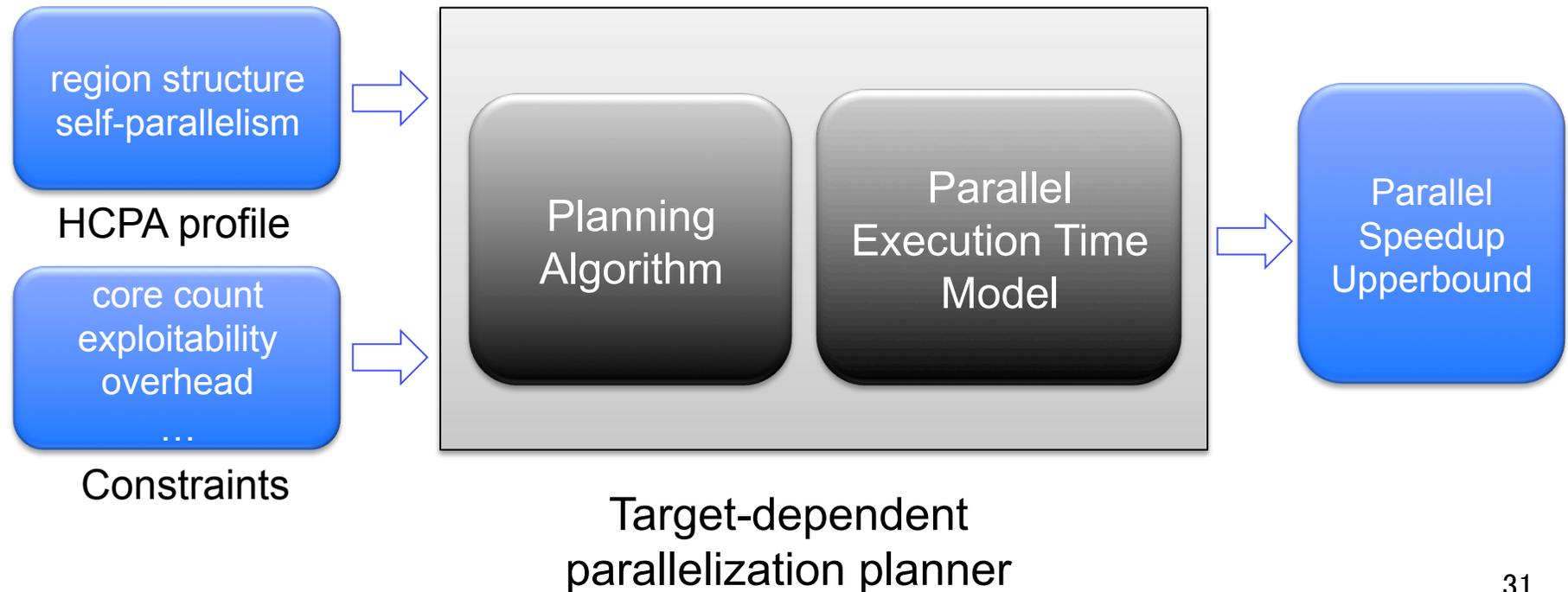
How many cores are available for parallelization?

Overhead

Do overheads eclipse the benefit of the parallelism?
e.g. scheduling, communication, synchronization

Parallelization Planner Overview

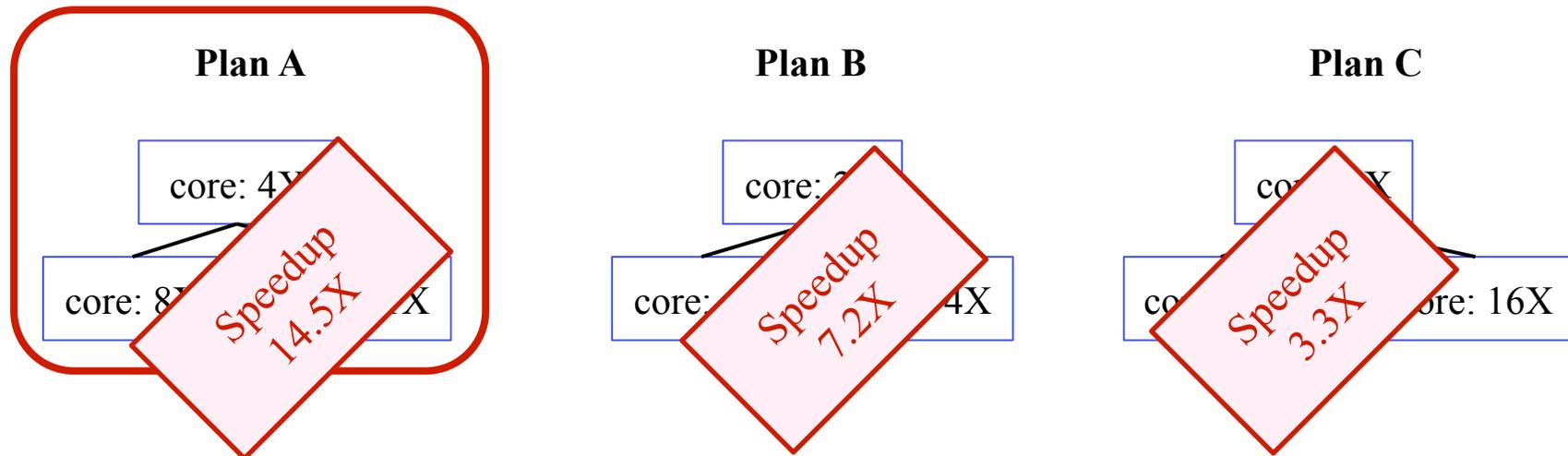
Goal: Find the speedup upperbound based on the HCPA results and parallelization constraints.



Planning Algorithm: Finding the Best Core Allocation

Estimate the execution time for each plan and pick the one with the highest speedup.

Highest Speedup



Core allocation plans for 32 cores

How can we evaluate the execution time for a specific core allocation?

Parallel Execution Time Model: A Bottom-up Approach

- Bottom-up evaluation with each region's estimated speedup and parallelization overhead $O(R)$

$$ptime(R) = \begin{cases} \frac{\sum_{k=1}^n ptime(child(R, k))}{speedup(R)} + O(R) & \text{R is a non-leaf region} \\ \frac{work(R)}{speedup(R)} + O(R) & \text{R is a leaf region} \end{cases}$$

`ptime(loop i)`

More Details in the Paper

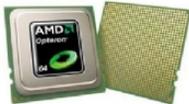
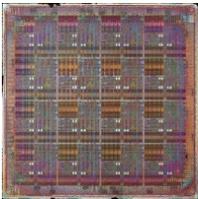
- How do we reduce the log file size of HCPA by orders of magnitude?
- What is the impact of exploitability in speedup estimation?
- How do we predict superlinear speedup?
- And many others...

Outline

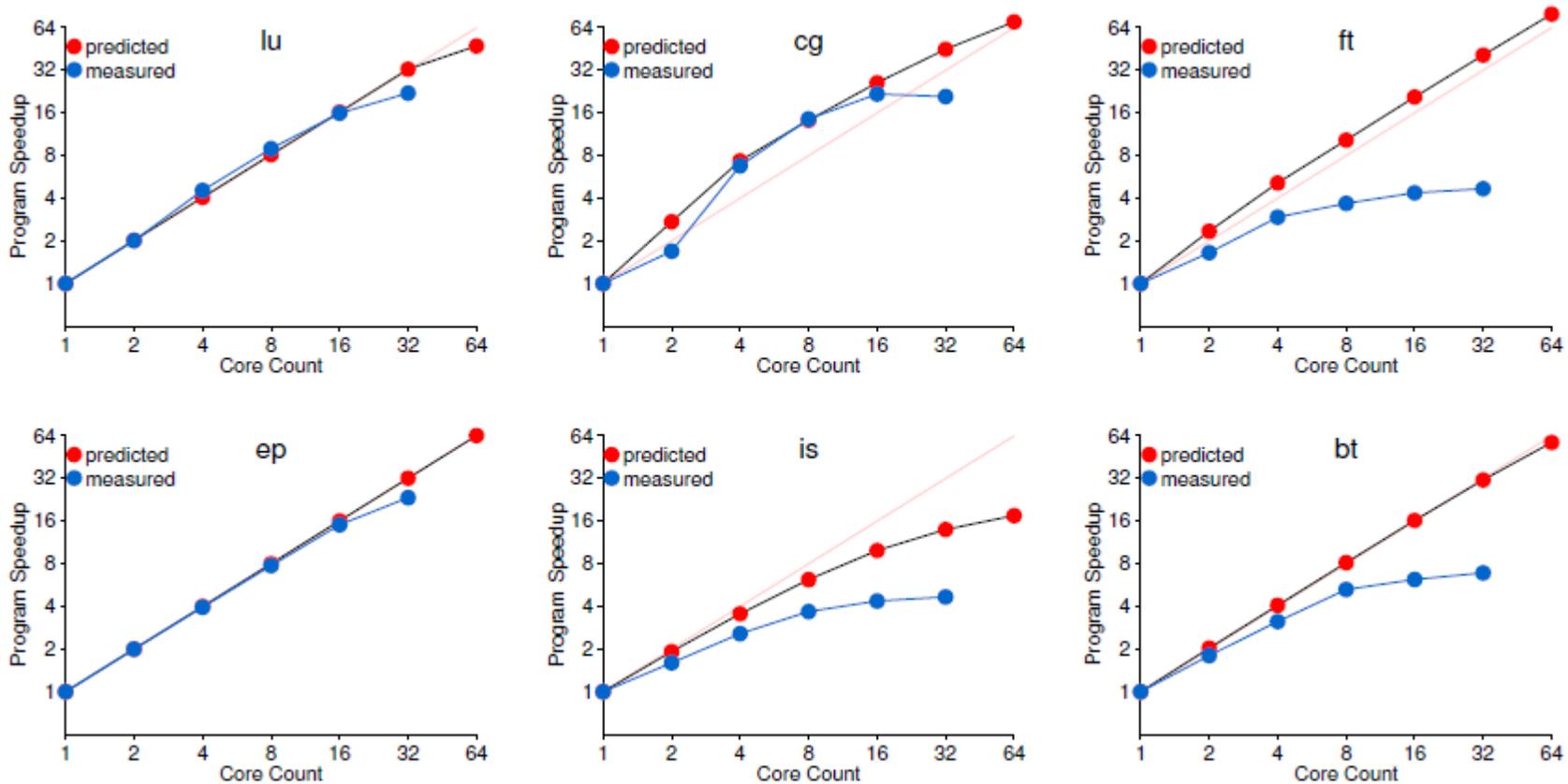
- Introduction
- Background: Critical Path Analysis
- How Kismet Works
- **Experimental Results**
- Conclusion

Methodology

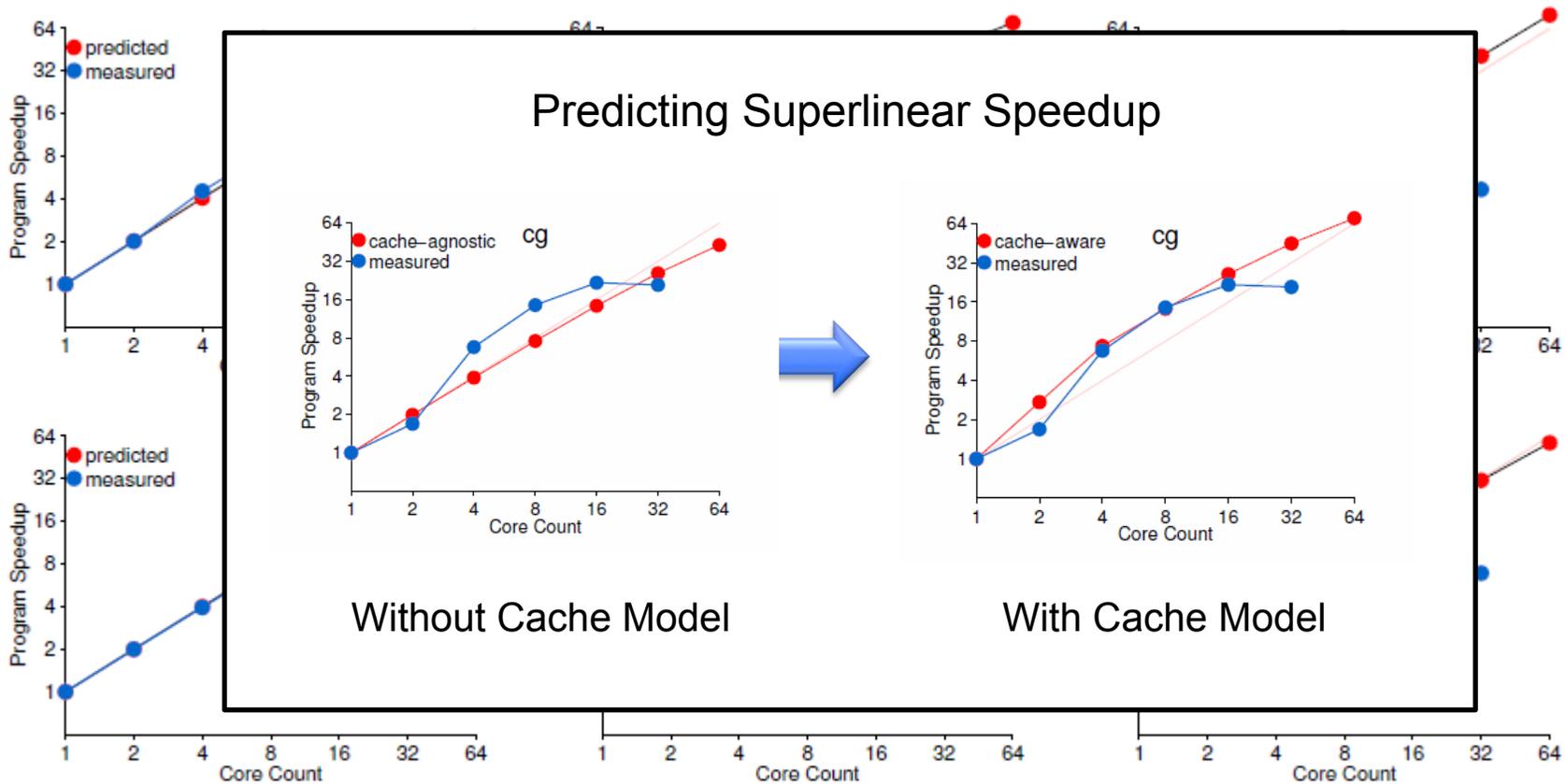
- Compare estimated and measured speedup
- To show Kismet's wide applicability, we targeted two very different platforms

Platform	 Multicore	 Raw
Processor	8 * Quad Core AMD Opteron 8380	16-core MIT Raw
Parallelization Method	OpenMP (Manual)	RawCC (Automatic)
Exploitable Parallelism	Loop-Level Parallelism (LLP)	Instruction-Level Parallelism (ILP)
Synchronization Overhead	High (> 10,000 cycles)	Low (< 100 cycles)

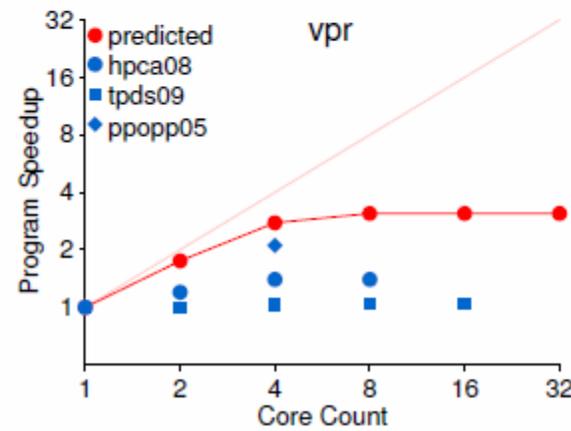
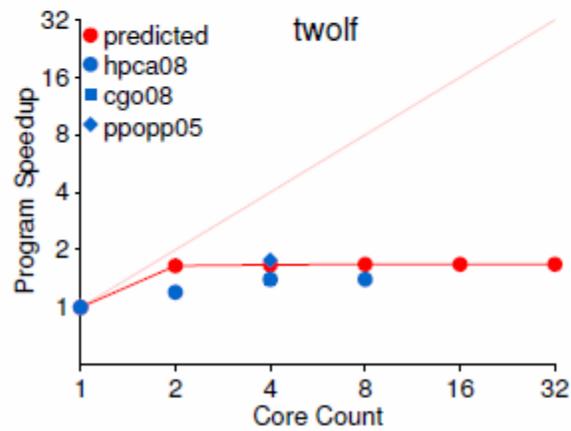
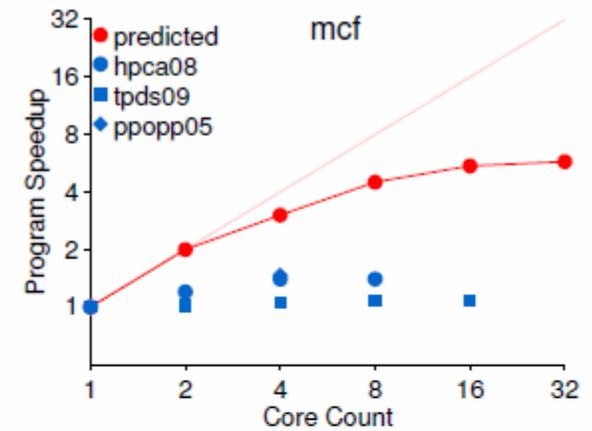
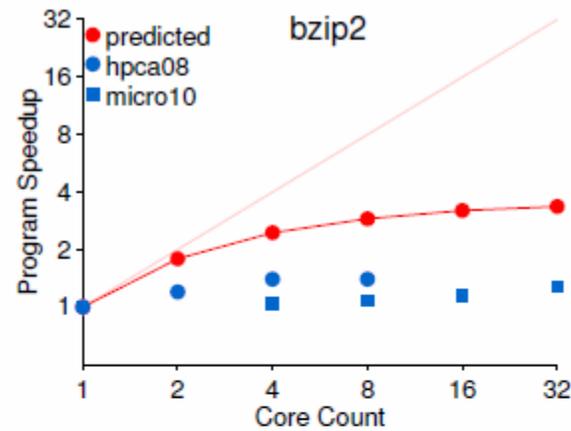
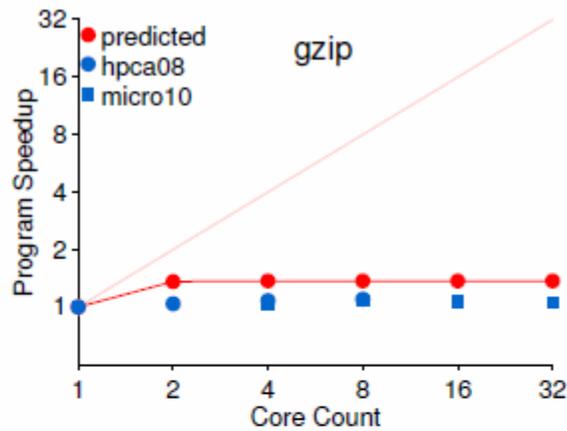
Speedup Upperbound Predictions: NAS Parallel Benchmarks



Speedup Upperbound Predictions: NAS Parallel Benchmarks



Speedup Upperbound Predictions: Low-Parallelism SpecInt Benchmarks



Conclusion



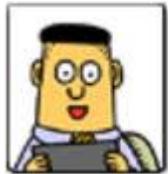
Kismet provides parallel speedup upperbound from serial source code.



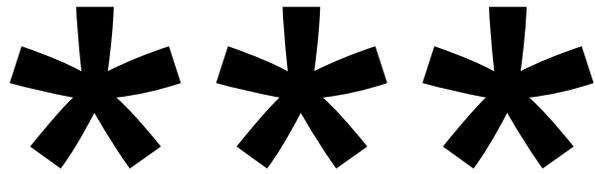
HCPA profiles self-parallelism using a hierarchical region model and the parallelization planner finds the best parallelization strategy.



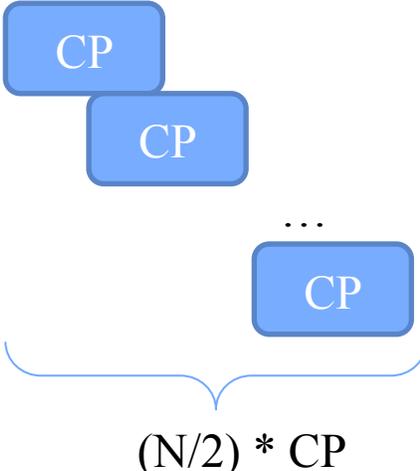
We demonstrated Kismet's ability to accurately estimate parallel speedup on two different platforms.



Kismet will be available for public download in the first quarter of 2012.

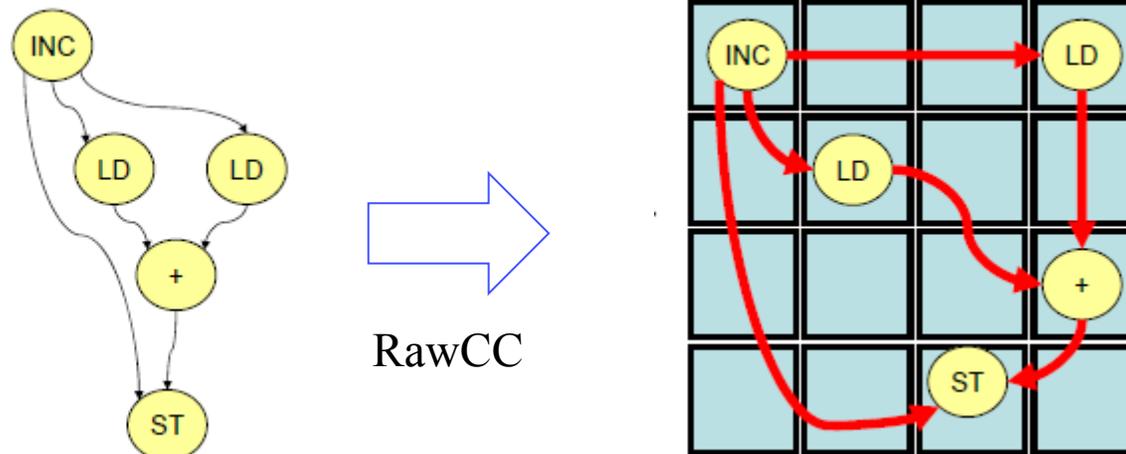


Self-Parallelism for Three Common Loop Types

Loop Type	DOALL	DOACROSS	Serial
Loop's Critical Path Length (cp)			
Work	$N * CP$	$N * CP$	$N * CP$
Self-Parallelism	$\frac{N * CP}{CP} = N$	$\frac{N * CP}{(N/2) * CP} = 2.0$	$\frac{N * CP}{N * CP} = 1.0$

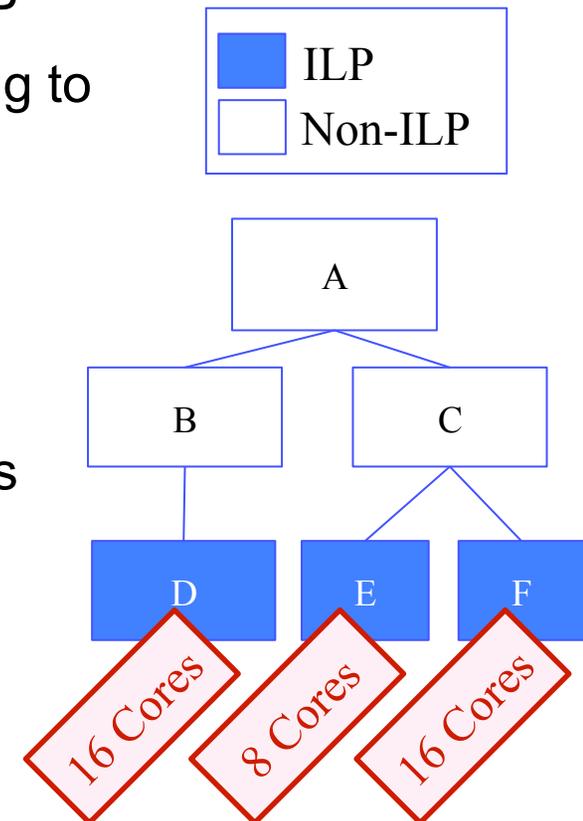
Raw Platform: Target Instruction-Level Parallelism

- Exploits ILP in each basic block by executing instructions on multiple cores
- Leverages a low-latency inter-core network to enable fine-grained parallelization
- Employs loop unrolling to increase ILP in a basic block

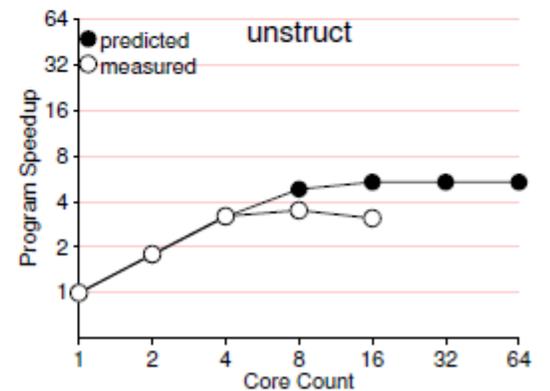
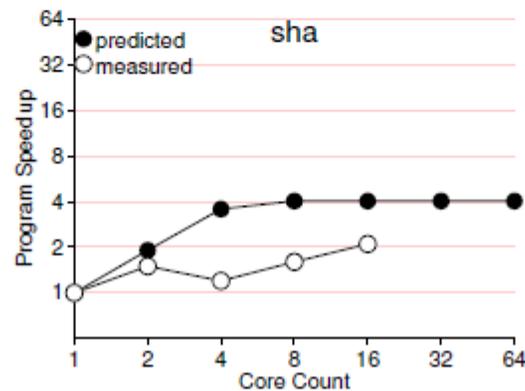
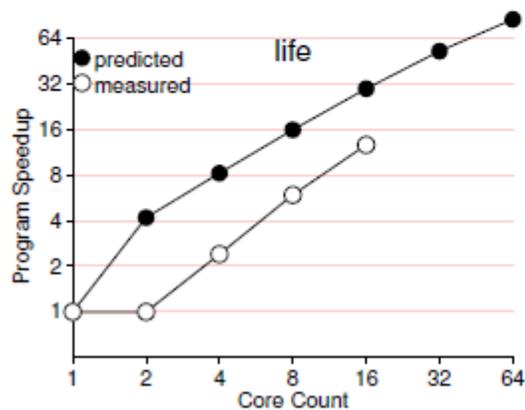
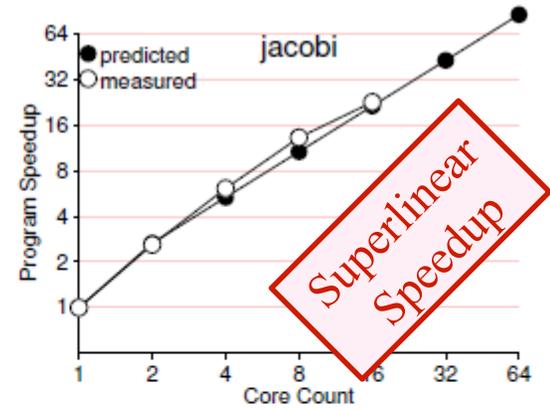
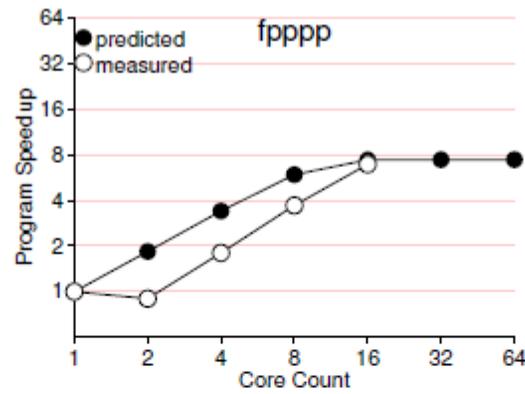
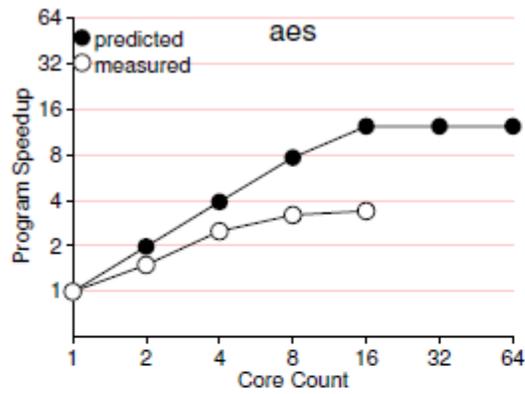


Adapting Kismet to Raw

- Constraints to filter unprofitable patterns
 - Target only leaf regions as they capture ILP
 - Like RawCC, Kismet performs loop unrolling to increase ILP, possibly bringing superlinear speedup
- Greedy Planning Algorithm
 - Greedy algorithm works well as leaf regions will run independent of each other
 - Parallelization overhead limits the optimal core count for each region

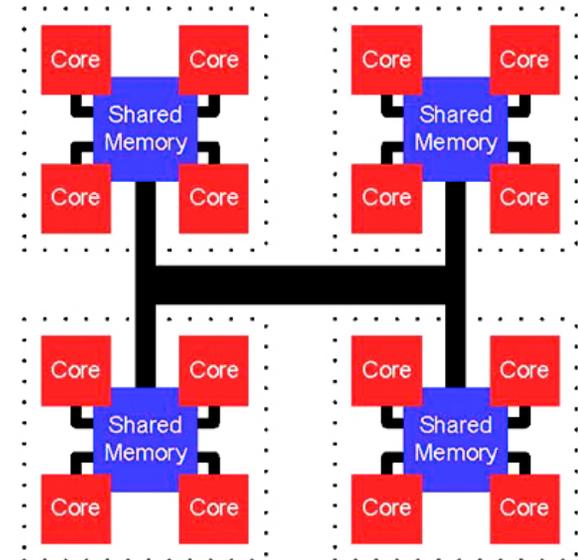


Speedup Upperbound Predictions: Raw Benchmarks



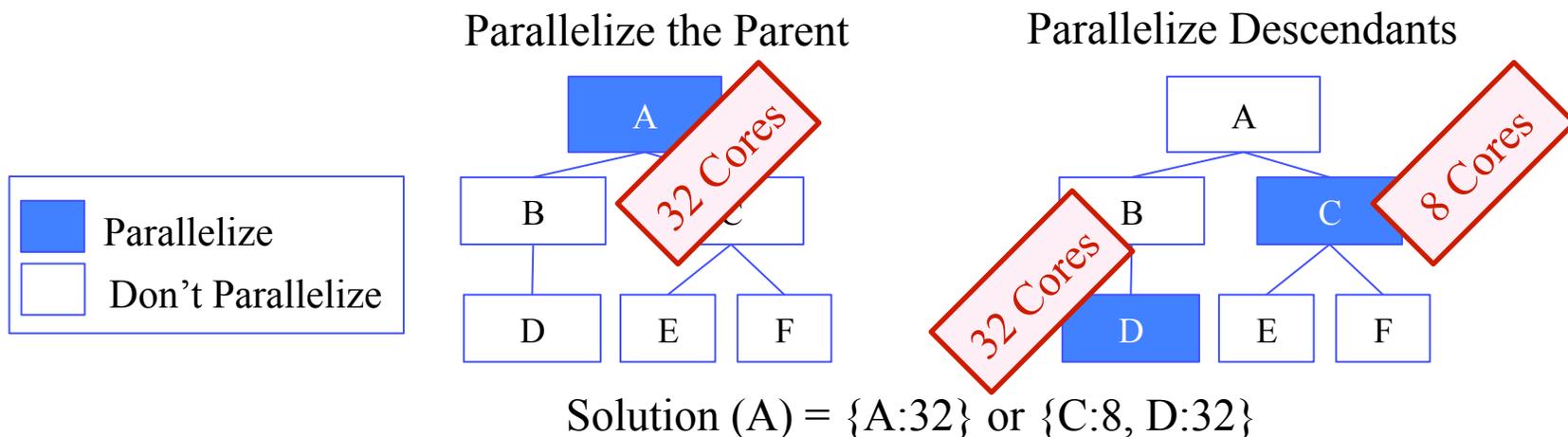
Multicore Platform: Target Loop-Level Parallelism

- Models OpenMP parallelization focusing on loop-level parallelism
- Disallows nested parallelization due to excessive synchronization overhead via shared memory
- Models cache effect to incorporate increased cache size from multiple cores



Adapting Kismet to Multicore

- Constraints to filter unprofitable OpenMP usage
 - Target only loop-level parallelism
 - Disallow nested parallelization
- Bottom-up Dynamic Programming
 - Parallelize either parent region or a set of descendants
 - Save the best parallelization for a region R in *Solution(R)*



Impact of Memory System

- Gather cache miss ratios for different cache sizes
- Log load / store counts for each region
- Integrate memory access time in time model

