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*.

1. Produce instrumented binary with *kismet-cc*
2. Perform parallelism profiling with a sample input
3. Estimate speedup under given constraints

```
$> make CC=kismet-cc

$> $(PROGRAM) $(INPUT)

$> kismet -opteron -openmp
Cores   1   2   4   8   16   32
Speedup 1   2   3.8  3.8  3.8  3.8 (est.)
```

Kismet’s easy-to-use usage model
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 upper bound 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?

```assembly
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?

```plaintext
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?

1a    $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

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 Path?

How Does CPA Compute Critical Path Path?

node: dynamic instruction with latency

dependence between instructions

work: serial execution time, total sum of node weights

critical path length (cp): minimum parallel execution time

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

Total-Parallelism = 1.33
Total-Parallelism Metric: Captures the Ideal Speedup of a Program

Min 1.0

Totally Serial

......

Total-Parallelism

All the work on the critical path

Highly Parallel

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?

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Measured Speedup (16 cores)</th>
<th>CPA Estimated Speedup</th>
<th>Optimism Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep</td>
<td>15.0</td>
<td>9722</td>
<td>648</td>
</tr>
<tr>
<td>life</td>
<td>12.6</td>
<td>116278</td>
<td>9228</td>
</tr>
<tr>
<td>is</td>
<td>4.4</td>
<td>1300216</td>
<td>295503</td>
</tr>
<tr>
<td>sp</td>
<td>4.0</td>
<td>189928</td>
<td>47482</td>
</tr>
<tr>
<td>unstruct</td>
<td>3.1</td>
<td>3447</td>
<td>1112</td>
</tr>
<tr>
<td>sha</td>
<td>2.1</td>
<td>4.8</td>
<td>2.3</td>
</tr>
</tbody>
</table>

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( )
{
    ....
ininner();
}

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

- **CPA**
- **Kismet**
  - **Hierarchical Critical Path Analysis**
    - Measure parallelism with a hierarchical region model
  - **Parallelization Planner**
    - Find the best parallelization strategy with target constraints
Revisiting CPA Problem #1: Data-flow Style Execution Model Is Unrealistic

```c
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

```
for (i=0 to 4)
  for (j=0 to 32)
    foo();
  for (k=0 to 2)
    bar1();
    bar2();
```

HCPA Step 1. Hierarchical Region Modeling
HCPA Step 2: Recursively Apply CPA

Total-Parallelism from \textit{inner()} = \sim 7X

Total-Parallelism from \textit{middle()} and \textit{inner()} = \sim 6X

Total-Parallelism from \textit{outer()}, \textit{middle()}, and \textit{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

```c
for (i=0 to 100) {
    a[i] = a[i] + 1;
    b[i] = b[i] - 1;
}
```

Total-Parallelism (from CPA)

```c
for (i=0 to 100) {
    a[i] = a[i] + 1;
    b[i] = b[i] - 1;
}
```

Self-Parallelism (from HCPA)
HCPA Step 3: Quantifying Self-Parallelism

Generalized Self-Parallelism Equation

For a non-leaf node $R$, $SP(R)$ is given by:

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

Self-P (inner) = ~7.0 X

Self-P (middle) = ~1.0 X

Self-P (outer) = ~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.

- Core count
- Exploitability
- Overhead
- Region structure
- Self-parallelism
- HCPA profile
- Constraints

Target-dependent parallelization planner

Planning Algorithm

Parallel Execution Time Model

Parallel Speedup Upperbound
Planning Algorithm: Allocates Cores with Key Constraints

for (i=0 to 4)
  for (j=0 to 32)
    foo ();
  for (k=0 to 2)
    bar1();
    bar2();

The allocated core count should not exceed $\text{cei} [\text{self-p}]$.

If a region’s parallelism is not exploitable, do not parallelize the region.

The product of allocated cores from the root to a leaf should not exceed the total available core count.
Planning Algorithm:
Finding the Best Core Allocation

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

**Highest Speedup**

<table>
<thead>
<tr>
<th>Plan A</th>
<th>Plan B</th>
<th>Plan C</th>
</tr>
</thead>
<tbody>
<tr>
<td>core: 4X</td>
<td>core: 2X</td>
<td>core: 8X</td>
</tr>
<tr>
<td>core: 8X</td>
<td>core: 4X</td>
<td>core: 16X</td>
</tr>
<tr>
<td>Speedup 14.5X</td>
<td>Speedup 7.2X</td>
<td>Speedup 3.3X</td>
</tr>
</tbody>
</table>

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 $O(R)$

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

$\text{ptime(loop \ i)}$
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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Multicore</th>
<th>Raw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>8 * Quad Core AMD Opteron 8380</td>
<td>16-core MIT Raw</td>
</tr>
<tr>
<td>Parallelization Method</td>
<td>OpenMP (Manual)</td>
<td>RawCC (Automatic)</td>
</tr>
<tr>
<td>Exploitable Parallelism</td>
<td>Loop-Level Parallelism (LLP)</td>
<td>Instruction-Level Parallelism (ILP)</td>
</tr>
<tr>
<td>Synchronization Overhead</td>
<td>High (&gt; 10,000 cycles)</td>
<td>Low (&lt; 100 cycles)</td>
</tr>
</tbody>
</table>
Speedup Upperbound Predictions: NAS Parallel Benchmarks
Speedup Upperbound Predictions: NAS Parallel Benchmarks

Predicting Superlinear Speedup

Without Cache Model  With Cache Model
Speedup Upperbound Predictions: Low-Parallelism SpecInt Benchmarks

- gzip
- bzip2
- mcf
- twolf
- vpr
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

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>DOALL</th>
<th>DOACROSS</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop’s</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Critical Path</td>
<td>CP</td>
<td>(N/2) * CP</td>
<td>CP</td>
</tr>
<tr>
<td>Length (cp)</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Work</td>
<td>N * CP</td>
<td>N * CP</td>
<td>N * CP</td>
</tr>
</tbody>
</table>
| Self-       | \[
| Parallelism | \frac{N \times CP}{CP} = N
|             | \frac{N \times CP}{(N/2) \times CP} = 2.0
|             | \frac{N \times CP}{N \times 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

Superlinear Speedup
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)

Solution (A) = \{A:32\} or \{C:8, D:32\}
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