An example of a research compiler

Simone Campanoni
simonec@eecs.northwestern.edu
Sequential programs are not accelerating like they used to.
Multicores are underutilized

**Single application:**
Not enough explicit parallelism
  - Developing parallel code is hard
  - Sequentially-designed code is still ubiquitous

**Multiple applications:**
Only a few CPU-intensive applications running concurrently in client devices
Parallelizing compiler: Exploit unused cores to accelerate sequential programs
Non-numerical programs need to be parallelized
Parallelize loops to parallelize a program

99% of time is spent in loops

Outermost loops
DOALL parallelism

Iteration 0

Iteration 1

Iteration 2

work()
DOACROSS parallelism

Sequential segment

Parallel segment

Time
HELIX: DOACROSS for multicore

HELIX: DOACROSS for multicore

HELIX: DOACROSS for multicore

HELIX: DOACROSS for multicore

HELIX: DOACROSS for multicore

Parallelize loops to parallelize a program

99% of time is spent in loops

Outermost loops

Innermost loops

99% of time is spent in loops

Time
Parallelize loops to parallelize a program

- **Innermost loops**
- **Outermost loops**

- **Coverage**
- **Ease of analysis**
- **Communication**

HELIX
HELIX: DOACROSS for multicore


Speedup

SPEC INT baseline

16 -

Innermost loops

Outermost loops

Coverage

Easy of analysis

Loop Parallelism

HELIX-RC

HELIX-UP

Small Loop Parallelism

4-core Intel Nehalem

1 -

4 -
Outline

Small Loop Parallelism and HELIX

[CGO 2012
DAC 2012,
IEEE Micro 2012]

HELIX-RC: Architecture/Compiler Co-Design
[ISCA 2014]

HELIX-UP: Unleash Parallelization
[CGO 2015]
SLP challenge: short loop iterations

Duration of loop iteration (cycles)

SPEC CPU
Int benchmarks
SLP challenge: short loop iterations

![Diagram showing the distribution of loop iteration durations. The x-axis represents the duration of loop iteration (cycles), and the y-axis represents the percentage of loop iterations. The diagram includes a shaded area and a dotted line indicating SPEC CPU Int benchmarks.]
SLP challenge: short loop iterations

![Graph showing percentage of loop iterations vs. duration of loop iteration (cycles). The graph compares different generations of processors: Ivy Bridge, Nehalem, and Atom. The x-axis represents the duration of loop iteration (cycles), and the y-axis represents the percentage of loop iterations. The red arrow indicates the duration where adjacent core communication latency is significant.]
A compiler-architecture co-design to efficiently execute short iterations

**Compiler**
- Identify latency-critical code in each small loop
  - Code that generates shared data
- Expose information to the architecture

**Architecture: Ring Cache**
- Reduce the communication latency on the critical path
Light-weight enhancement of today’s multicore architecture

Store X, 1
Store Y, 1
Iter. 0

Core

Core 0

Core 1

Ring node

Ring node

DL1

DL1

Last level cache

75 – 260 cycles!

Store X, 1

Store Y, 1
Iter. 0

Load X

Load Y
Iter. 1

Iter. 2

Core 2

Core 3

Iter. 3
Light-weight enhancement of today’s multicore architecture

Store X, 1
Wait 0
Store Y, 1
Signal 0
Iter. 0

Core 0

Ring node

Core 1

Ring node

... Wait 0
Load Y
... Iter. 1

Iter. 0

Store X, 1
Wait 0
Store Y, 1
Signal 0
Iter. 0
Simulator: XIOSim, DRAMSim
Compiler : ILDJIT (LLVM)

Latency: 1 cycle
Bandwidth: 70 bits for signals
68 bits for data

98% hit rate
The importance of HELIX-RC
The importance of HELIX-RC

![Bar chart showing program speedup comparison between HELIX and HELIX-RC for both numerical and non-numerical programs. The x-axis represents different programs, and the y-axis represents the program speedup. The chart indicates a significant improvement in speedup for HELIX-RC compared to HELIX, especially noticeable in numerical programs.]
Outline

Small Loop Parallelism and HELIX

HELIX-RC: Architecture/Compiler Co-Design

HELIX-UP: Unleash Parallelization

[CGO 2012
DAC 2012,
IEEE Micro 2012]

[ISCA 2014]

[CGO 2015]
HELIX and its limitations

Thread 0 • Thread 1 • Thread 2 • Thread 3

Iteration 0 • Data
Iteration 1 • Data
Iteration 2 • Data

Performance:
• Lower than you would like
• Inconsistent across architectures
• Sensitive to dependence analysis accuracy

What can we do to improve it?

4 Cores

Nehalem: 78% accuracy, 1.19
Bulldozer: 79% accuracy, 1.61
Haswell: 80% accuracy, 2.31

80% → 50%
Opportunity: relax program semantics

- Some workloads tolerate output distortion

- Output distortion is workload-dependent
Relaxing transformations remove performance bottlenecks

• Sequential bottleneck

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 1</td>
<td>Inst 1</td>
<td>Inst 1</td>
</tr>
<tr>
<td>Inst 2</td>
<td>Inst 2</td>
<td>Inst 2</td>
</tr>
<tr>
<td>Inst 3</td>
<td>Dep</td>
<td>Inst 3</td>
</tr>
<tr>
<td>Inst 4</td>
<td>Dep</td>
<td>Inst 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inst 3</td>
</tr>
<tr>
<td>Speedup</td>
<td></td>
<td>Inst 4</td>
</tr>
</tbody>
</table>

Sequential segment
Relaxing transformations remove performance bottlenecks

• Sequential bottleneck

• Communication bottleneck

• Data locality bottleneck
Relaxing transformations remove performance bottlenecks

No relaxing transformations
Relaxing transformation 1
Relaxing transformation 2
...
Relaxing transformation k

No output distortion
Baseline performance

Max output distortion
Max performance
Design space of HELIX-UP

1) User provides output distortion limits
2) System finds the best configuration
3) Run parallelized code with that configuration
Pruning the design space

**Empirical observation:**
Transforming a code region affects only the loop it belongs to

50 loops, 2 code regions per loop
2 transformations per code region

Complete space \( = 2^{100} \)
Pruned space \( = 50 \times (2^2) = 200 \)

How well does HELIX-UP perform?
HELIX-UP unblocks extra parallelism with small output distortions

HELIX: no relaxing transformations

Program Speedup

Hardware threads

12
11
10
9
8
7
6
5
4
3
2
1
0

177.mesa 179.art 183.equake 256.bzip2 blackscholes swaptions Geomean

Output distortion: 3.8%

Nehalem 6 cores
2 threads per core
HELIX-UP unblocks extra parallelism with small output distortions

Nehalem 6 cores
2 threads per core
Performance/distortion tradeoff

256.bzip2

Normalized Performance

% Output Distortion

Static HELIX-UP
Run time code tuning

• Static HELIX-UP decides how to transform the code based on profile data averaged over inputs

• The runtime reacts to transient bottlenecks by adjusting code accordingly
Adapting code at run time unlocks more parallelism

256.bzip2

Normalized Performance

% Output Distortion

Static HELIX-UP
HELIX-UP improves more than just performance

• Robustness to DDG inaccuracies

• Consistent performance across platforms
Relaxed transformations to be robust to DDG inaccuracies

Increasing DDG inaccuracies leads to lower performance

No impact on HELIX-UP
Relaxed transformations for consistent performance

Increasing communication latency

Output distortion: 0.7% 0.75% 1%
Small Loop Parallelism and HELIX
• *Parallelism hides in small loops*

HELIX-RC: Architecture/Compiler Co-Design
• *Irregular programs require low latency*

HELIX-UP: Unleash Parallelization
• *Tolerating distortions boosts parallelization*
Thank you!
Small Loop Parallelism and HELIX
• *Parallelism hides in small loops*

HELIX-RC: Architecture/Compiler Co-Design
• *Irregular programs require low latency*

HELIX-UP: Unleash Parallelization
• *Tolerating distortions boosts parallelization*