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
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

Time
Parallelize loops to parallelize a program

- Innermost loops
- Outermost loops

Coverage
Ease of analysis
Communication

HELIX
HELIX: DOACROSS for multicore


Innermost loops

Outermost loops

Coverage

Easy of analysis

Loop Parallelism

Communication

HELIX-RC
HELIX-UP

16-core Intel Nehalem

16

4

1

Speedup

SPEC INT baseline

ICC, Microsoft Visual Studio, DOACROSS
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

SPEC CPU Int benchmarks
SLP challenge: short loop iterations

- Duration of loop iteration (cycles)
- Percentage of loop iterations

Adjacent core communication latency

- Nehalem
- Ivy Bridge
- Atom
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

Iter. 0
Store X, 1
Store Y, 1

Iter. 1
Core 0
Core 1
Core 2
Core 3

Ring node
Ring node
Ring node

DL1
DL1
DL1

Last level cache

Iter. 2
Iter. 3

Store X, 1
Store Y, 1

Load X
Load Y

... Iter. 1

75 - 260 cycles!
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

Core 0
Ring node

Core 1
Ring node
Simulator: XIOSim, DRAMSim
Compiler: ILDJIT (LLVM)

98% hit rate

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

16 Intel Atom
1.6 GHz

2 cycles

1 KB Ring Node

1 cycle

32 KB DL1 Cache

Last Level Cache

Size 8 MB
The importance of HELIX-RC

![Graph showing program speedup]
The importance of HELIX-RC

![Bar chart comparing HELIX and HELIX-RC for non-numerical and numerical programs]
Outline

Small Loop Parallelism and HELIX

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

HELIX-UP: Unleash Parallelization
[CGO 2015]
HELIX and its limitations

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

What can we do to improve it?
Opportunity: relax program semantics

• Some workloads tolerate output distortion

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

- Sequential bottleneck
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

- 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

![Graph showing performance distortion tradeoff with 256.bzip2 data and HELIX compared to 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

% 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%

Normalized Performance

HELIX on Nehalem
HELIX on Bulldozer
HELIX on Haswell
UP on Nehalem
UP on Bulldozer
UP on Haswell
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*