HELIX-RC
An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs

Simone Campanoni, Kevin Brownell, Svilen Kanev
Timothy M. Jones, Gu-Yeon Wei, David Brooks

HARVARD
School of Engineering and Applied Sciences
Today’s commodity platforms include multiple cores

Use multiple cores for a single program

Distribute loop iterations among cores
Today’s commodity platforms include multiple cores
Motivation

Today's commodity platforms include multiple cores
Today’s commodity platforms include multiple cores
Today’s commodity platforms include multiple cores
Motivation

Today's commodity platforms include multiple cores.
Motivation

Today's commodity platforms include multiple cores

CGO 2012, HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing

Program Speedup

ICC, Visual studio w/ parallelization

State of the art

Program Speedup

1x

2x
Today’s commodity platforms include multiple cores

Program Speedup

ICC, Visual studio w/ parallelization

State of the art

ICC -O3

1x

2x
Motivation

Today's commodity platforms include multiple cores.
Today’s commodity platforms include multiple cores

Ring Cache makes automatic parallelization practical for conventional multicores

- State of the art: 2x
- HELIX-RC: 7x

Program Speedup
Outline

- Opportunity of small loops
- The HELIX-RC solution
- Evaluation of HELIX-RC
Outline

- Opportunity of small loops
- The HELIX-RC solution
- Evaluation of HELIX-RC
Opportunity

Code complexity
- Control flow
- Data flow
Opportunity

Code complexity
- Control flow
- Data flow

Dependences to satisfy

Prior Works

Thread-Level-Speculation (TLS)

HELIX-RC targets small (hot) loops

No TLS

Enable code transformations to recompute shared values
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works

- Thread-Level-Speculation (TLS)
  - ↓ Apparent
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - ↓ Apparent
- TLS overhead ⇒ big loops
Opportunity

- Code complexity
  - Control flow
  - Data flow

- Dependences to satisfy
  - Actual
  - Apparent

Prior Works

- Thread-Level-Speculation (TLS)
  - Apparent
- TLS overhead ⇒ big loops (10× more dependences!)
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works

- Thread-Level-Speculation (TLS)
  - ↓ Apparent
- TLS overhead ⇒ big loops (10× more dependences!)
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - ↓ Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC targets small (hot) loops
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC targets small (hot) loops
- Code complexity ↓
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC targets small (hot) loops
- Code complexity
  - Apparent (only 1.2× more dependences)
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - ↓ Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC targets small (hot) loops
- No TLS
  - ↓ Code complexity
  - ↓ Apparent (only 1.2× more dependences)
## Opportunity

<table>
<thead>
<tr>
<th>Code complexity</th>
<th>Dependences to satisfy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>Actual</td>
</tr>
<tr>
<td>Data flow</td>
<td>Apparent</td>
</tr>
</tbody>
</table>

### Prior Works

- **Thread-Level-Speculation (TLS)**
  - Apparent
  - TLS overhead ⇒ big loops (10× more dependences!)

### HELIX-RC targets small (hot) loops

- Code complexity
  - Apparent (only 1.2× more dependences))
- Enable code transformations to recompute shared values

No TLS
Opportunity

↑ Code complexity
- Control flow
- Data flow

Dependences to satisfy ↑
- Actual
- Apparent ↑

Prior Works
- Thread-Level-Speculation (TLS)
  - ↓ Apparent
- TLS overhead ⇒ big loops (10× more dependences!)

HELIX-RC targets small (hot) loops
- No TLS
  - ↓ Code complexity
    - ↓ Apparent (only 1.2× more dependences))
  - Enable code transformations to recompute shared values
    - ↓ Actual
Main Challenge for Small Loops

Very short loop iterations
Main Challenge for Small Loops

Very short loop iterations

![Graph showing percentage of loop iterations vs clock cycles]
Main Challenge for Small Loops

Very short loop iterations
Main Challenge for Small Loops

Very short loop iterations

- Measured cache coherence latency
- Nehalem
- Ivy Bridge
- Atom

Clock Cycles

Percentage of loop iterations
Main Challenge for Small Loops

Very short loop iterations

Graph showing the percentage of loop iterations as a function of clock cycles. The graph indicates that very short loop iterations can lead to significant cache coherence latency, especially on Nehalem and Ivy Bridge architectures. The percentage of loop iterations increases sharply before reaching a plateau, highlighting the challenge of optimizing small loops in modern CPU architectures.
Main Challenge for Small Loops

Very short loop iterations

![Graph showing percentage of loop iterations versus clock cycles for different processors (Ivy Bridge, Nehalem, Atom). The graph indicates a sharp increase in percentage of loop iterations as the clock cycles increase, especially around 25 clock cycles, highlighting the main challenge for small loops.](image-url)
Outline

- Opportunity of small loops
- The HELIX-RC solution
- Evaluation of HELIX-RC
Split the Work Among Compiler and Architecture

Compiler: HCCv3

Architecture: Ring Cache
Split the Work Among Compiler and Architecture

Compiler: HCCv3
- Identify code that *may* generate shared data

Architecture: Ring Cache
Split the Work Among Compiler and Architecture

Compiler: HCCv3
- Identify code that *may* generate shared data
- Expose information to architecture

Architecture: Ring Cache
Split the Work Among Compiler and Architecture

**Compiler: HCCv3**
- Identify code that *may* generate shared data
- Expose information to architecture

**Architecture: Ring Cache**
Drastically reduce the communication latency
Split the Work Among Compiler and Architecture

**Compiler: HCCv3**
- Identify code that *may* generate shared data
- Expose information to architecture

**Architecture: Ring Cache**
Drastically reduce the communication latency

---

Traditional coherence protocol

0

75
Ivy
Bridge
Split the Work Among Compiler and Architecture

Compiler: HCCv3
- Identify code that *may* generate shared data
- Expose information to architecture

Architecture: Ring Cache
Drastically reduce the communication latency
- Proactive data distribution

HELIX-RC

Traditional coherence protocol

0

Ivy
Bridge
Split the Work Among Compiler and Architecture

Compiler: HCCv3
- Identify code that *may* generate shared data
- Expose information to architecture

Architecture: Ring Cache
- Drastically reduce the communication latency
- Proactive data distribution

**HELIX-RC**

Traditional coherence protocol

Program Speedup

- 2x
- 7x

Ivy Bridge
Split the Work Among Compiler and Architecture

**Compiler: HCCv3**
- Identify code that *may* generate shared data
- Expose information to architecture

**Architecture: Ring Cache**
- Drastically reduce the communication latency
- Proactive data distribution

**HELIX-RC**

Traditional coherence protocol

![Program Speedup](2x-7x)
1. Parallelize most promising loops
   - Identify code that *may* generate shared loop iteration data
   - Keep shared data in memory
1. Parallelize most promising loops
   - Identify code that
     *may* generate
     shared loop iteration data
   - Keep shared data in memory
1. Parallelize most promising loops
   - Identify code that may generate shared loop iteration data
   - Keep shared data in memory
Compiler HCCv3

1. Parallelize most promising loops
   - Identify code that *may* generate shared loop iteration data
   - Keep shared data in memory
1. Parallelize most promising loops
   - Identify code that *may* generate shared loop iteration data
   - Keep shared data in memory
1. Parallelize most promising loops
   - Identify code that may generate shared loop iteration data
   - Keep shared data in memory
1. Parallelize most promising loops
   - Identify code that *may* generate shared loop iteration data
   - Keep shared data in memory
Parallelism Among Sequential Segments

for (...){
    1: a = f(a);
    2: b = f(b);
}

- Loop-carried data dependences

- A small loop
for (...) {
    1: a = f(a);
    2: b = f(b);
}

Loop-carried data dependences

Sequential segments
Parallelism Among Sequential Segments

for (...) {
    1: a = f(a);
    2: b = f(b);
}

Sequential segments may generate shared data
Parallelism Among Sequential Segments

```c
for (...) {
    1: a = f(a);
    2: b = f(b);
}
```

- Sequential segments *may* generate shared data
  - Each sequential segment executes in loop-iteration order
Parallelism Among Sequential Segments

Sequential segments may generate shared data

- Each sequential segment executes in loop-iteration order
- Parallelism among sequential segments

```c
for (...) {
    1: a = f(a);
    2: b = f(b);
}
```
Small Loops Do Not Work On Commodity Multicore

```plaintext
for (...) {
    1: a = f(a);
    2: b = f(b);
}
```

Bottleneck
- Data movement

[Diagram showing loop-carried data dependences and sequential segments for Core 0 and Core 1 across iterations 0, 1, and 2.]
Small Loops Do Not Work On Commodity Multicore

for (...) {
    1: a = f(a);
    2: b = f(b);
}

Core 0

1

2

Iteration 0

Core 1

1

2

Bottleneck

- Data movement

Loop-carried data dependences

Sequential segments

---

Bottleneck

Measured cache coherence latency

Clock Cycles

0 25 50 75 100 125 150 175 200 225 250 275 300

0 25 50 75 100 150 200 250 300

Atom

Nehalem

Ivy Bridge

3x
Small Loops Do Not Work On Commodity Multicore

for (...) {
   1: a = f(a);
   2: b = f(b);
}

Bottleneck
- Data movement

Cut communication latency!
Cut communication latency!
Light Enhancement in Conventional Multicore Architecture
Light Enhancement in Conventional Multicore Architecture

Diagram showing the architecture with cores, DL1 cache, and last level cache.

SW: 0xB, 3
Light Enhancement in Conventional Multicore Architecture
Ring nodes cache shared data
Light Enhancement in Conventional Multicore Architecture

Accelerate communication shaped by the compiler
Light Enhancement in Conventional Multicore Architecture
Light Enhancement in Conventional Multicore Architecture
Light Enhancement in Conventional Multicore Architecture

- Core
  - Ring node
  - DL1 Cache
  - Last Level Cache
  - DL1 Cache
  - Ring node
  - Core

Data
Signals

\[ \text{sw } 0xA, 5 \]

[wait 1]

[sw 0XB, 3]

[signal 1] Sequential segment
Light Enhancement in Conventional Multicore Architecture

Sequential segment:
- sw 0xA,5
- wait 1
- sw 0XB,3
- signal 1
Light Enhancement in Conventional Multicore Architecture

Diagram of a conventional multicore architecture showing cores, ring nodes, and cache levels. The diagram includes data and signal paths with annotations indicating operations such as `sw 0xA,5`, `wait 1`, `sw 0XB,3`, and `signal 1`. The diagram also denotes a sequential segment.
Light Enhancement in Conventional Multicore Architecture

Delayed broadcast for unknown consumers

sw 0xA, 5
wait 1
sw 0XB, 3
signal 1
Sequential segment
Light Enhancement in Conventional Multicore Architecture

![Diagram of multicore architecture with light enhancement](Image)

- Core
  - Ring node
  - DL1 Cache
  - Last Level Cache
  - DL1 Cache
  - Ring node
  - Core

- Data
  - Signals

--sw 0xA, 5
- wait 1
- sw 0xB, 3
- signal 1

Sequential segment
Light Enhancement in Conventional Multicore Architecture

Sequential segment

Data
Signals

Core
Ring node
DL1 Cache
Last Level Cache
DL1 Cache
Ring node
Core

sw 0xA,5
wait 1
sw 0XB,3
signal 1
Light Enhancement in Conventional Multicore Architecture

```
sw 0xA,5
wait 1
sw 0XB,3
signal 1

Sequential segment
```
Core synchronization

No communication cost!
Access to remote data locally

No communication cost!
Architecture Parameters

Simulator: XIOSim, DRAMSim
Compiler: ILDJIT (LLVM)

- 16 Intel Atom
- 1.6 GHz
- 2 cycles
- 1 KB Ring Node
- 32 KB DL1 Cache
- Late Level Cache: Size 8 MB
- Latency: 1 cycle
- Bandwidth: 70 bits for signals, 68 bits for data

["XIOSim: Power-performance Modeling of Mobile x86 Cores"
ISLPED 2012, Svilen Kanev et al."]
Compiler-architecture co-design is effective for non-numerical workloads.
Compiler-architecture co-design is effective for *non-numerical* workloads

3 < speedup < 12
Compiler-architecture co-design is effective for *non-numerical* workloads

- $3 < \text{speedup} < 12$
- No slowdown!
The Importance of the Co-Design

- Compiler-architecture co-design is effective for *non-numerical* workloads
  - $3 < \text{speedup} < 12$
  - No slowdown!
Ring Cache vs. Ring Register
Ring Cache vs. Ring Register
The Importance of a Cache-Based Scheme

![Bar chart showing program speedup for different benchmarks using HELIX-RC and Ring Register.](chart.png)
Ring Cache Parameter Analysis

16 cores

Ring Node

Latency: 1 cycle

Bandwidth: 70 bits for signals

Last Level Cache
Size 8 MB

2 cycles

1 cycle

32 KB DL1 Cache

Unbounded 1 KB 32 KB 256 B

1 cycle 2 50

Unbounded 4 16 32

Unbounded 2 4 1

Programming Schedules

Benefits of decoupling specifications

Disrupting multi-threading, communication
Disrupting reg, cache, and memory access
Disrupting reg and memory access
Disrupting reg, cache, and communication

1 cycle 16 25

1 cycle 16 32

1 cycle 16 25

1 cycle 16 32

1 cycle 16 32

1 cycle 16 32
Core Parameters Analysis

- 16 cores
- 2 cycles
- 1 KB Ring Node
- 1 cycle
- 32 KB DL1 Cache
- Last Level Cache: Size 8 MB

Latency: 1 cycle
Bandwidth: 70 bits for signals
Ring Cache
makes
automatic parallelization practical
for conventional multicores
HELIX-RC

- Small loops $\Rightarrow$ few frequent dependences
  - Cut communication latency
  - Proactive data forwarding $\Rightarrow$ $\sim$0 communication latency
Conclusion

HELIX-RC

- Small loops $\Rightarrow$ few frequent dependences
  - Cut communication latency
  - Proactive data forwarding $\Rightarrow$ $\sim$0 communication latency

Questions?

http://helix.eecs.harvard.edu
Conclusion

We will release binaries for both http://helix.eecs.harvard.edu
We will release binaries for both 
http://helix.eecs.harvard.edu
Conclusion

We will release binaries for both http://helix.eecs.harvard.edu
Conclusion

We will release binaries for both

http://helix.eecs.harvard.edu