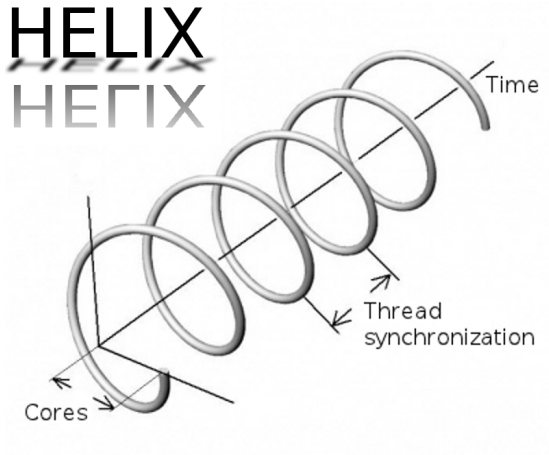
HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing

Simone Campanoni, Timothy M. Jones, Glenn Holloway
Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks



Summary

- Motivation
- A simple idea
- Single loop parallelization
- Loop selection
- Evaluation
- Conclusion

Extraction of Thread-Level-Parallelism (TLP)

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

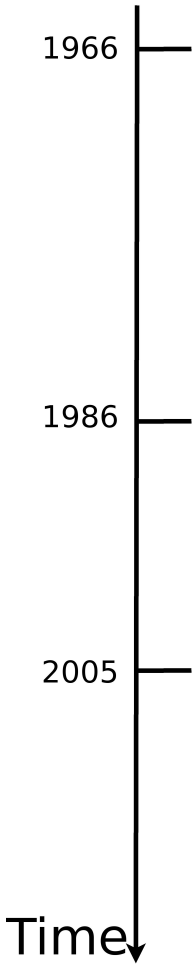
Main automatic approaches proposed:

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

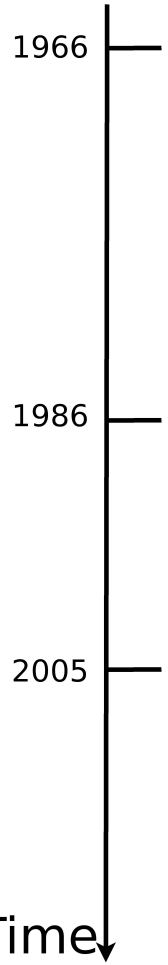


Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:



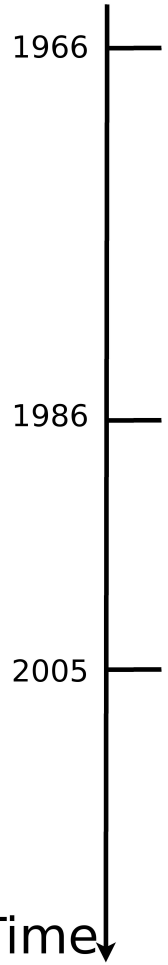
DOALL

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:



DOALL

Speedup increases with number of cores

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

2005

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

2005

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

Applicable to a broader set of programs

2005

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005

DSWP

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005

DSWP

Speedup are stable on inter-core communication delay

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966

DOALL

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986

DOACROSS

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005

DSWP

Speedup are stable on inter-core communication delay

Hard to predict speedup

Time

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966 — **DOALL**

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986 — **DOACROSS**

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005 — **DSWP**

Speedup are stable on inter-core communication delay

Hard to predict speedup

- Hard to avoid slowdown

Time ↓

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

1966 — **DOALL**

Speedup increases with number of cores

Limited applicability

- Loop-carried dependences not handled

1986 — **DOACROSS**

Applicable to a broader set of programs

Extremely sensitive to inter-core communication

2005 — **DSWP**

Speedup are stable on inter-core communication delay

Hard to predict speedup

- Hard to avoid slowdown

Time ↓

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

Is there a way to achieve all of these?

Speedup increases with number of cores

Applicable to a broader set of programs

Speedup are stable on inter-core communication delay

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

Is there a way to achieve all of these?

Speedup increases with number of cores

Applicable to a broader set of programs

Speedup are stable on inter-core communication delay

Produce predictable speedup

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

Applicable to a broader set of programs

Speedup are stable on inter-core communication delay

Produce predictable speedup

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

Speedup are stable on inter-core communication delay

Produce predictable speedup

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

Speedup are stable on inter-core communication delay

Produce speedup predictable enough to avoid slowdown

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

DOACROSS $<$ Stability of speedup $<$ DSWP

Produce speedup predictable enough to avoid slowdown

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

DOACROSS $<$ Stability of speedup $<$ DSWP

Produce speedup predictable enough to avoid slowdown

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

DOACROSS $<$ Stability of speedup $<$ DSWP

Inter-core communication $\Rightarrow$

Produce speedup predictable enough to avoid slowdown

Motivation

Extraction of Thread-Level-Parallelism (TLP)

- In multicore era: $\Uparrow$ performance $\Leftrightarrow$ TLP $\Uparrow$
- Manual approach: $\Uparrow$ software development time

Main automatic approaches proposed:

HELIX

Speedup increases with number of cores

General purpose technique

DOACROSS $<$ Stability of speedup $<$ DSWP

Inter-core communication $\Rightarrow$ private cache access hit

Produce speedup predictable enough to avoid slowdown

Motivation (2)

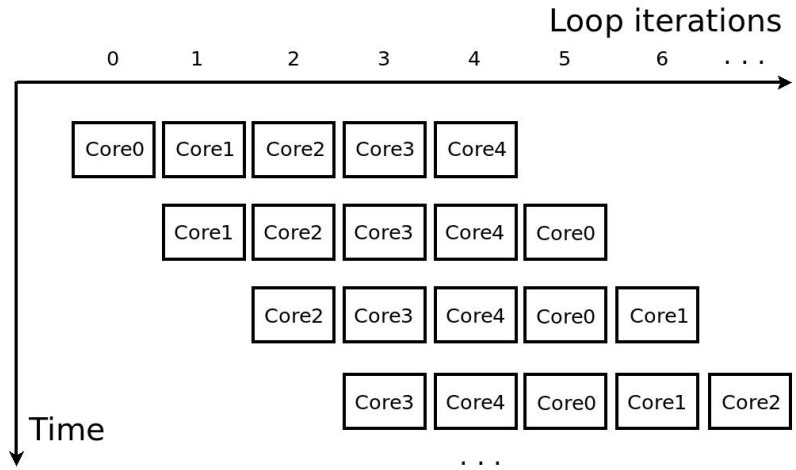
HELIX

- General purpose technique
- Predictable speedup
  - Avoid slowdown
- $|\text{threads}| \leq |\text{loop iterations}|$

Motivation (2)

HELIX

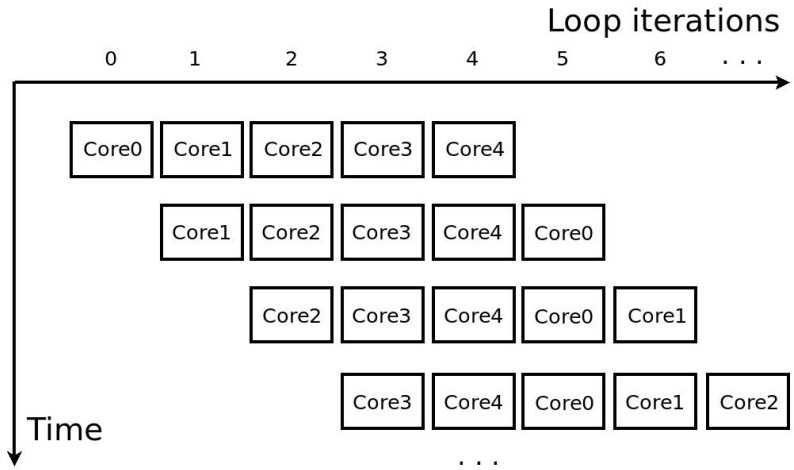
- General purpose technique
- Predictable speedup
  - Avoid slowdown
- $|\text{threads}| \leq |\text{loop iterations}|$
  - TLP extracted between loop iterations
  - Iterations grouped on modular value



Motivation (2)

HELIX

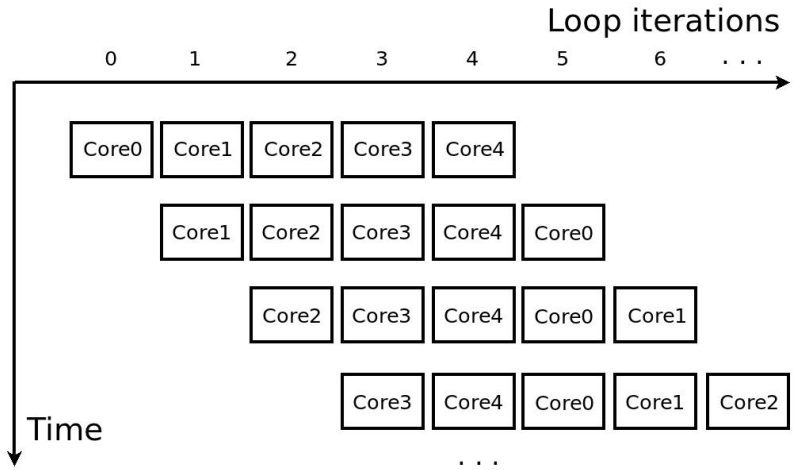
- General purpose technique
- Predictable speedup
  - Avoid slowdown
- $|\text{threads}| \leq |\text{loop iterations}|$
  - TLP extracted between loop iterations
  - Iterations grouped on modular value
- Automatic selection of loops



Motivation (2)

HELIX

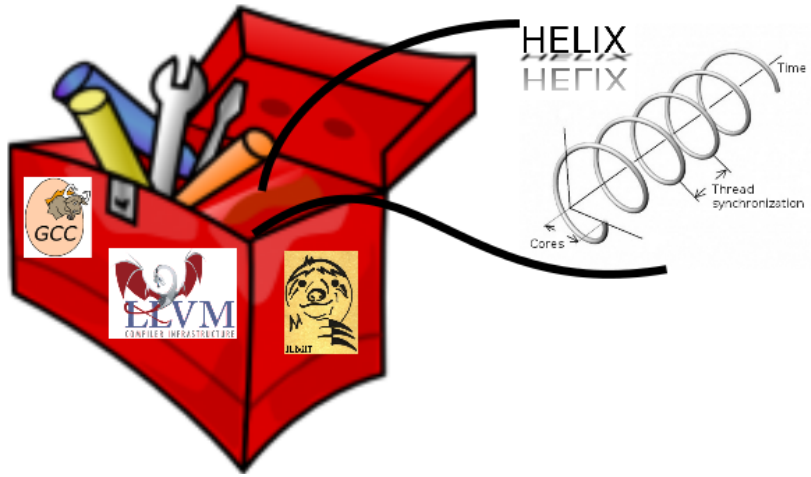
- General purpose technique
- Predictable speedup
  - Avoid slowdown
- $|\text{threads}| \leq |\text{loop iterations}|$
  - TLP extracted between loop iterations
  - Iterations grouped on modular value
- Automatic selection of loops
- *Easy* to implement



Motivation (2)

HELIX

- General purpose technique
- Predictable speedup
  - Avoid slowdown
- $|\text{threads}| \leq |\text{loop iterations}|$
  - TLP extracted between loop iterations
  - Iterations grouped on modular value
- Automatic selection of loops
- *Easy* to implement



Summary

- Motivation
- A simple idea
- Single loop parallelization
- Loop selection
- Evaluation
- Conclusion

A Simple Idea

```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```

- A simple program

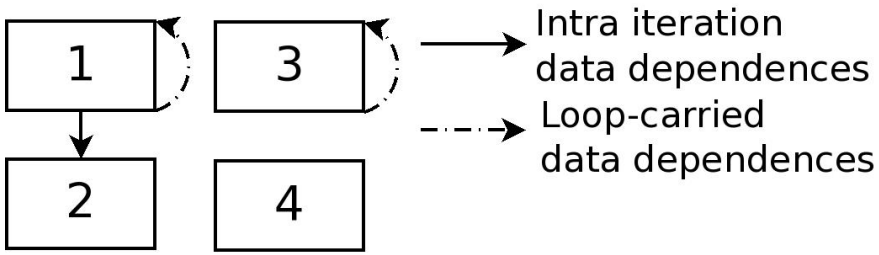
A Simple Idea

```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```

- Loop-carried data dependences 

A Simple Idea

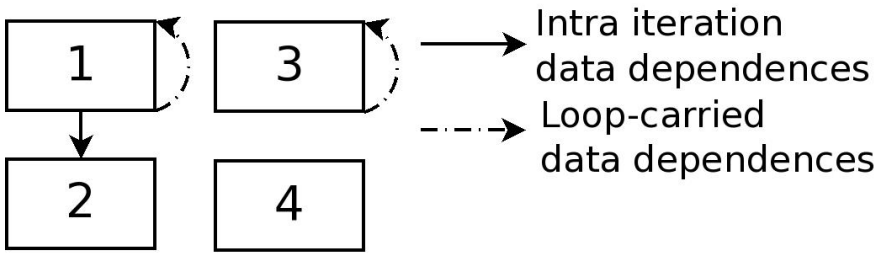
```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```



- Idea: exploit independent instructions

A Simple Idea

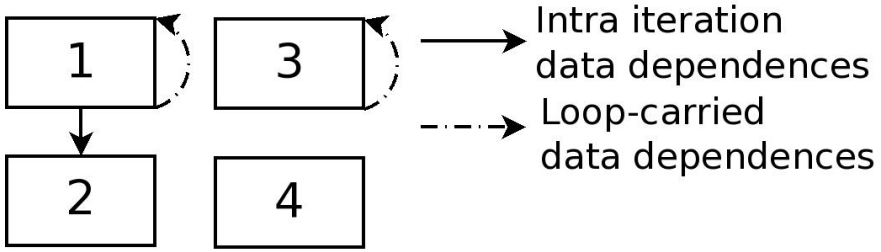
```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```

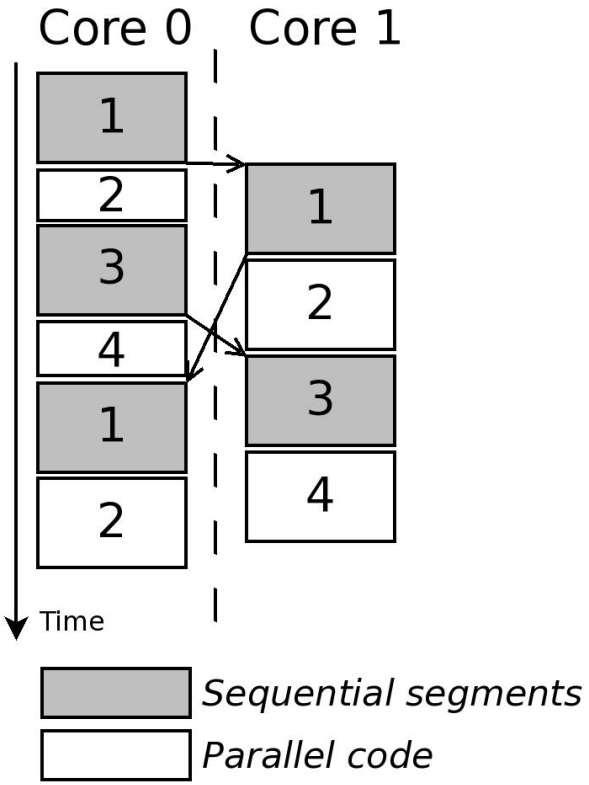


- Idea: exploit independent instructions *and*

A Simple Idea

```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```

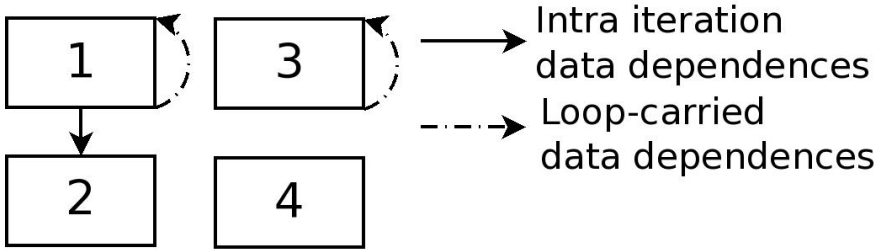


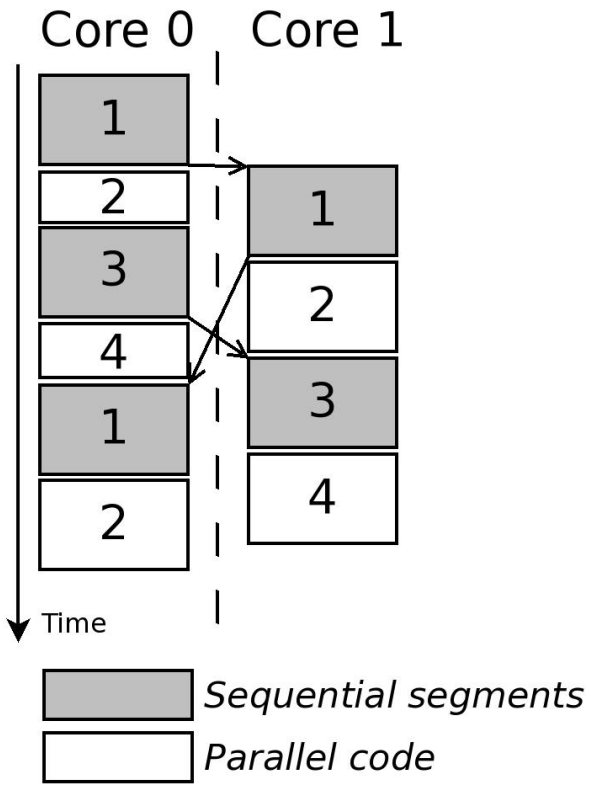


- Idea: exploit independent instructions *and* parallelism among sequential segments

A Simple Idea

```
for (...){
    1: a = update(a);
    2: work1(a);
    3: b = update(b);
    4: work2();
}
```





- Idea: exploit independent instructions *and* parallelism among sequential segments

Problem: amount of synchronization required increases drastically!

Overhead

Optimizations

Signalling

Notify threads

Adopted solutions

Overhead

Signalling

Notify threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send

Overhead

Signalling

Notify threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel
- Automatic selection of loops

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel
- Automatic selection of loops

Approach

- Select loops to parallelize

HELIX

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel
- Automatic selection of loops

Approach

- Select loops to parallelize
  - Light profile based selection

HELIX

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel
- Automatic selection of loops

Approach

- Select loops to parallelize
  - Light profile based selection
- Parallelize one loop at a time

Overhead

Signalling

Notify threads

Sequential code

Code that must execute in loop-iteration order

Data forwarding

Forward data between threads

Optimizations

Adopted solutions

- New code analysis to reduce the number of signals to send
- Code scheduling and use of SMT to reduce the delay per signal
- Code scheduling
- Execution of $\neq$ segments in parallel
- Automatic selection of loops

Approach

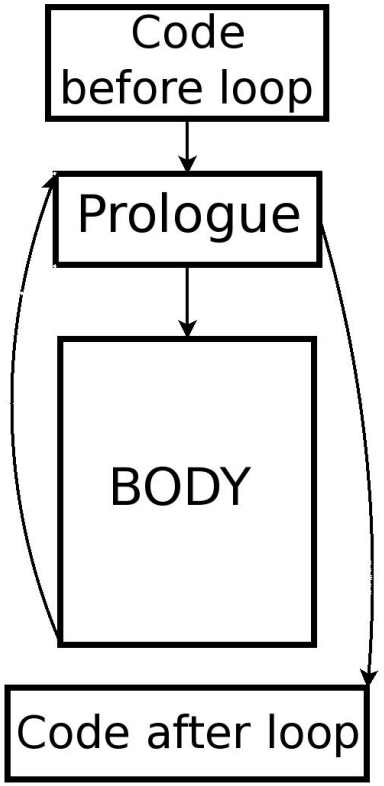
- Select loops to parallelize
  - Light profile based selection
- Parallelize one loop at a time
  - Each loop uses all cores decided at compile time

Summary

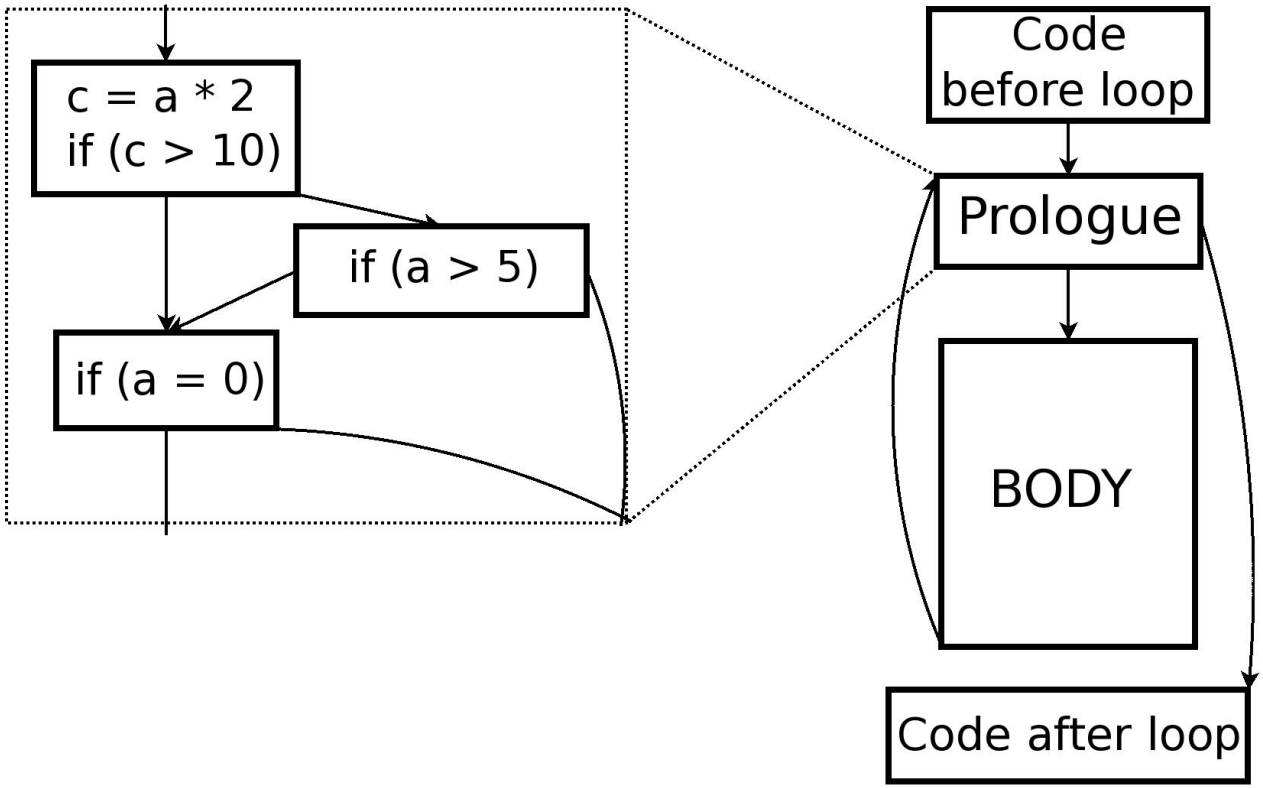
- Motivation
- A simple idea
- Single loop parallelization
- Loop selection
- Evaluation
- Conclusion

Step 1: Normalizing the Loop

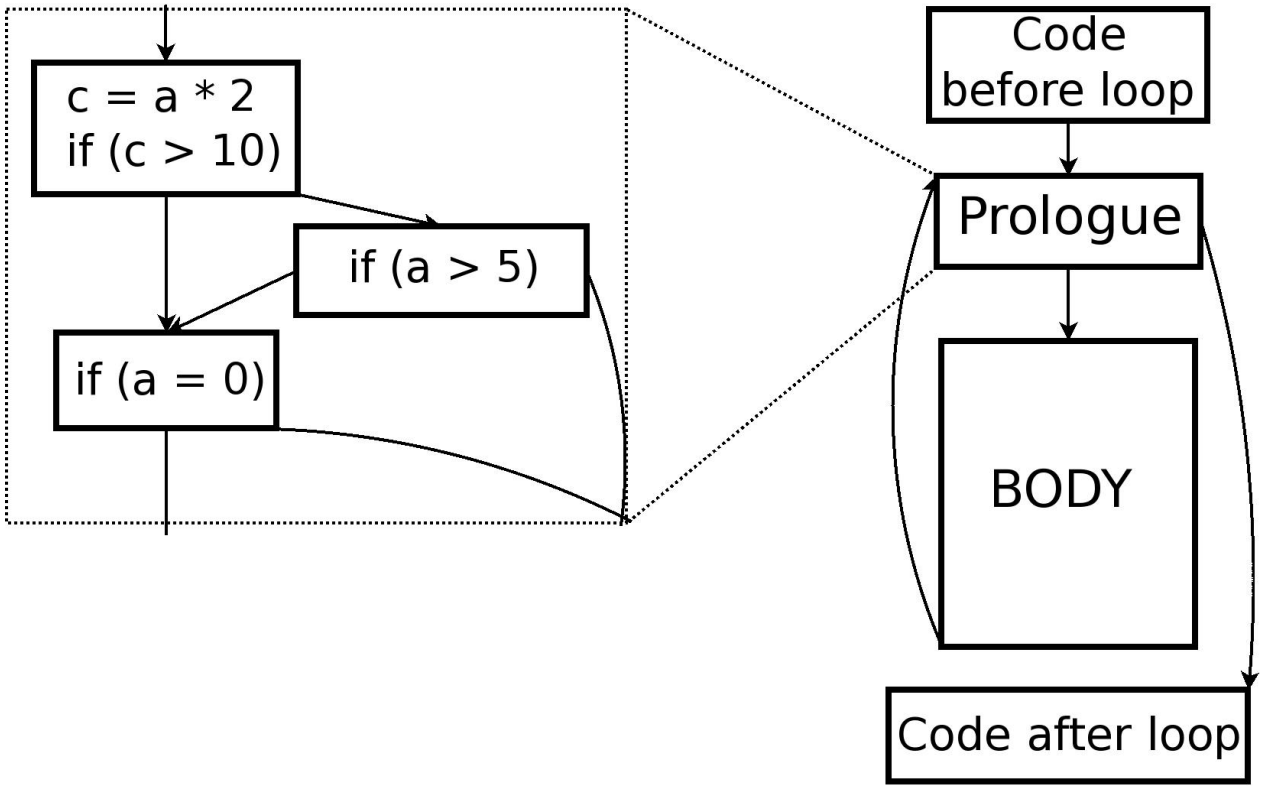
Step 1: Normalizing the Loop



Step 1: Normalizing the Loop

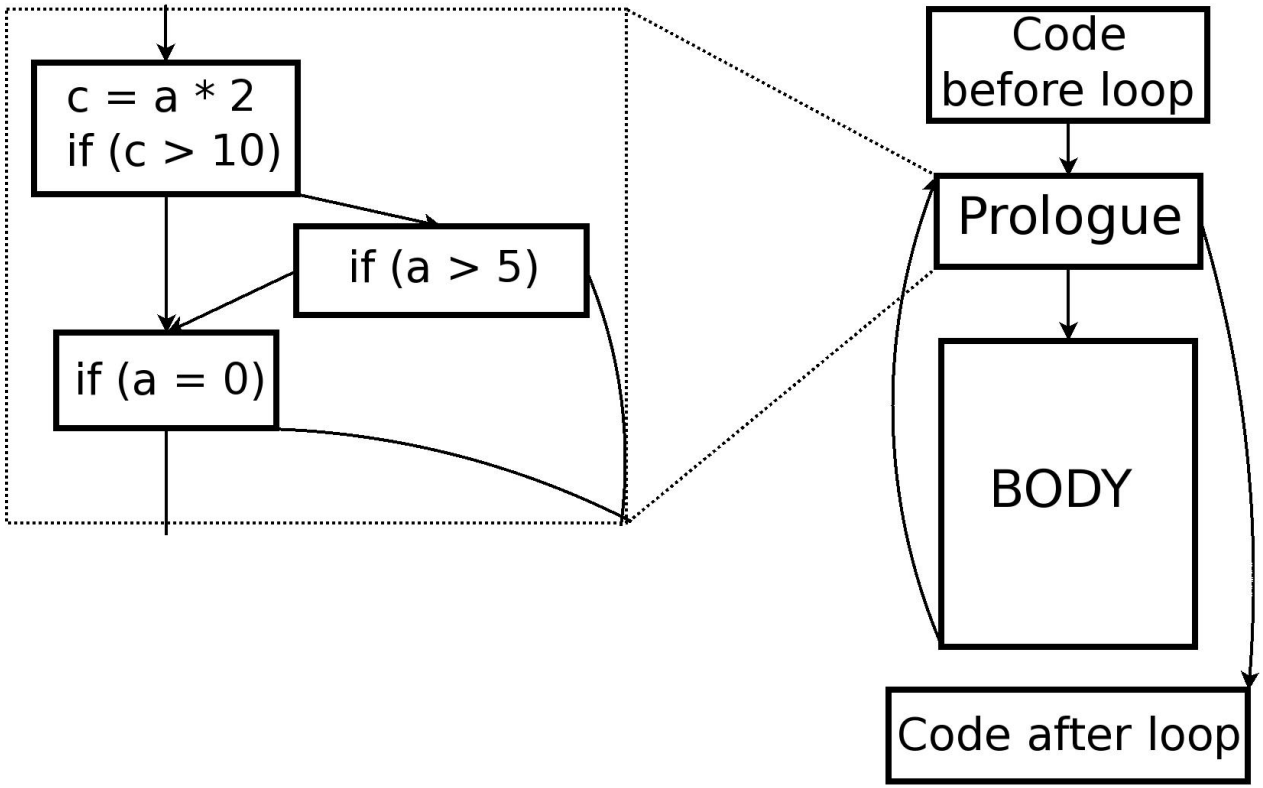


Step 1: Normalizing the Loop



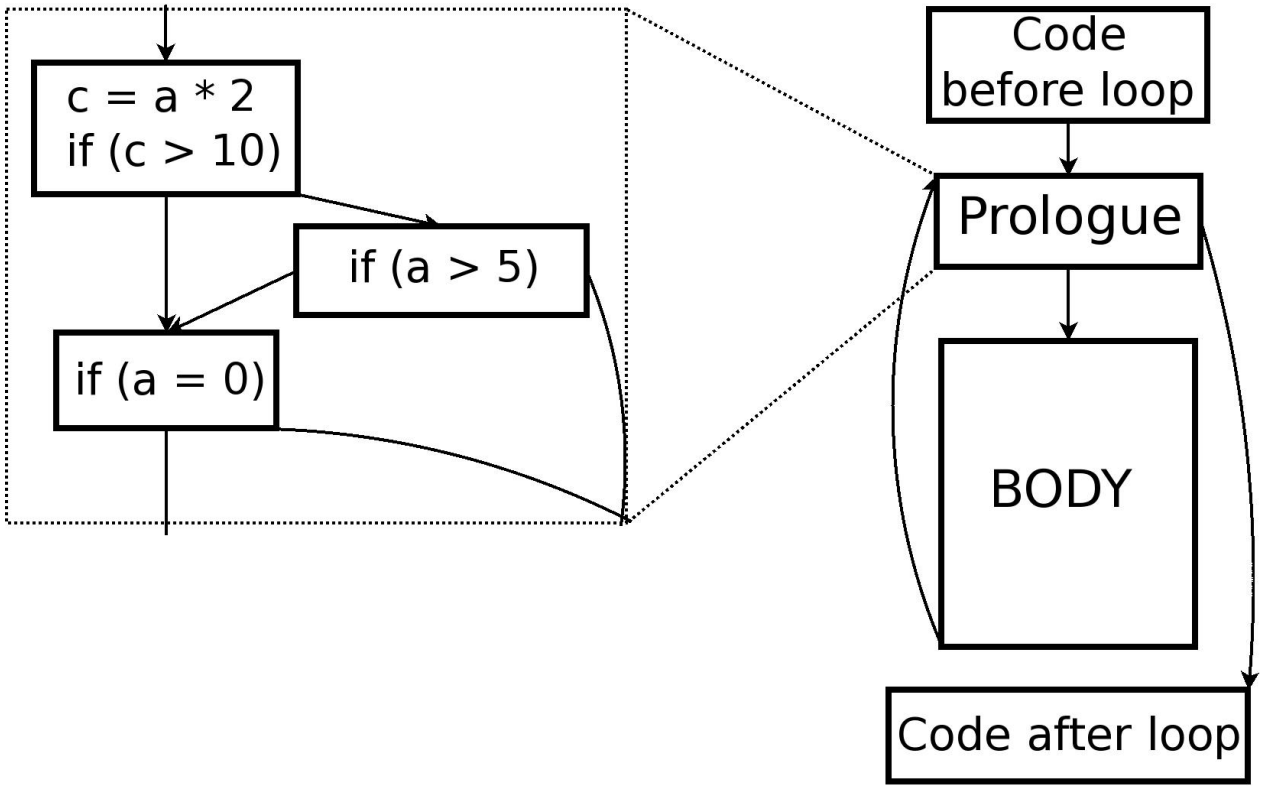
- The code is scheduled to minimize time spent $\in$ prologue
  - Reason: prologue is executed in loop-iteration order
- Best case: single exit controlled by an induction variable

Step 2: Identifying data dependences to satisfy



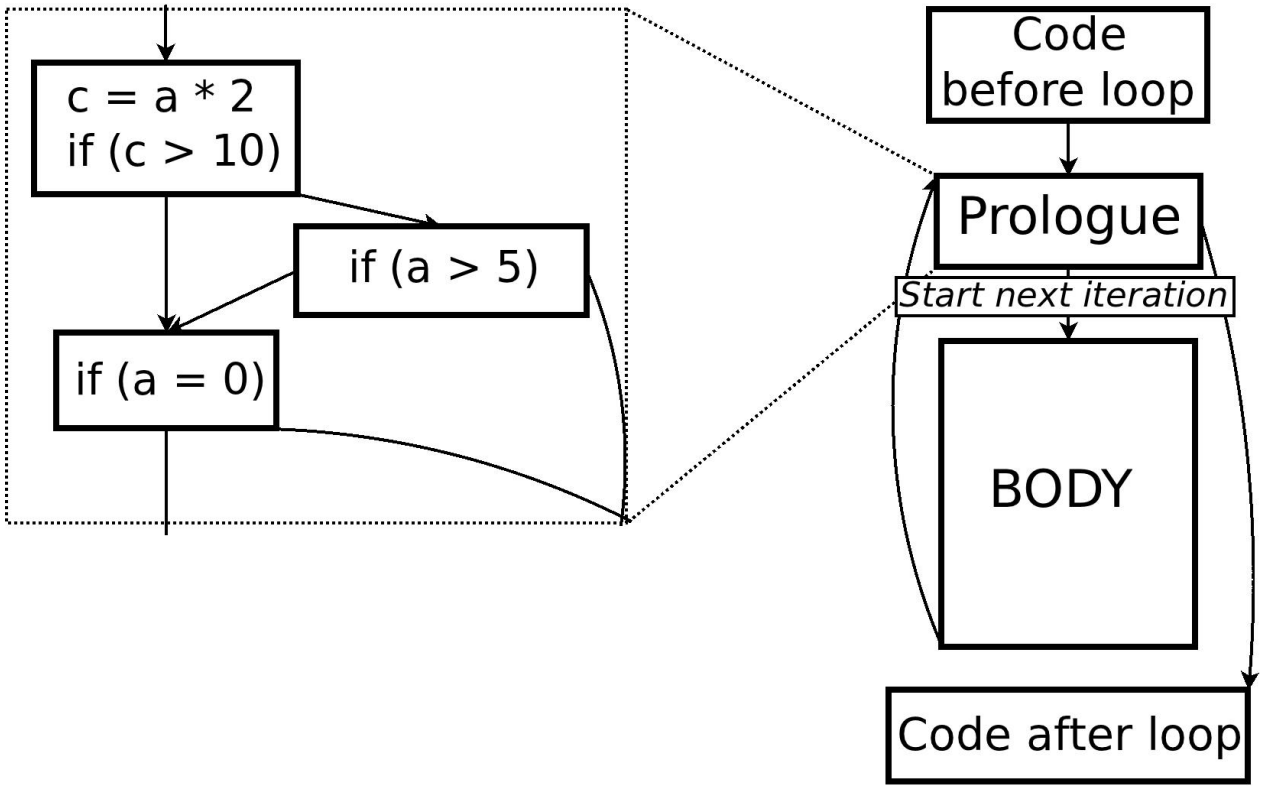
- The code is scheduled to minimize time spent $\in$ prologue
  - Reason: prologue is executed in loop-iteration order
- Best case: single exit controlled by an induction variable

Step 3: Starting next iterations



- The code is scheduled to minimize time spent $\in$ prologue
  - Reason: prologue is executed in loop-iteration order
- Best case: single exit controlled by an induction variable

Step 3: Starting next iterations



- The code is scheduled to minimize time spent $\in$ prologue
  - Reason: prologue is executed in loop-iteration order
- Best case: single exit controlled by an induction variable

Step 4: Computing Sequential Segments

Step 4: Computing Sequential Segments

For every $d = (a, b) \in D_{Data}$:

-
-

Step 4: Computing Sequential Segments

For every $d = (a, b) \in D_{Data}$:

- Instructions $Wait(d)$ are inserted as late as possible
-

Step 4: Computing Sequential Segments

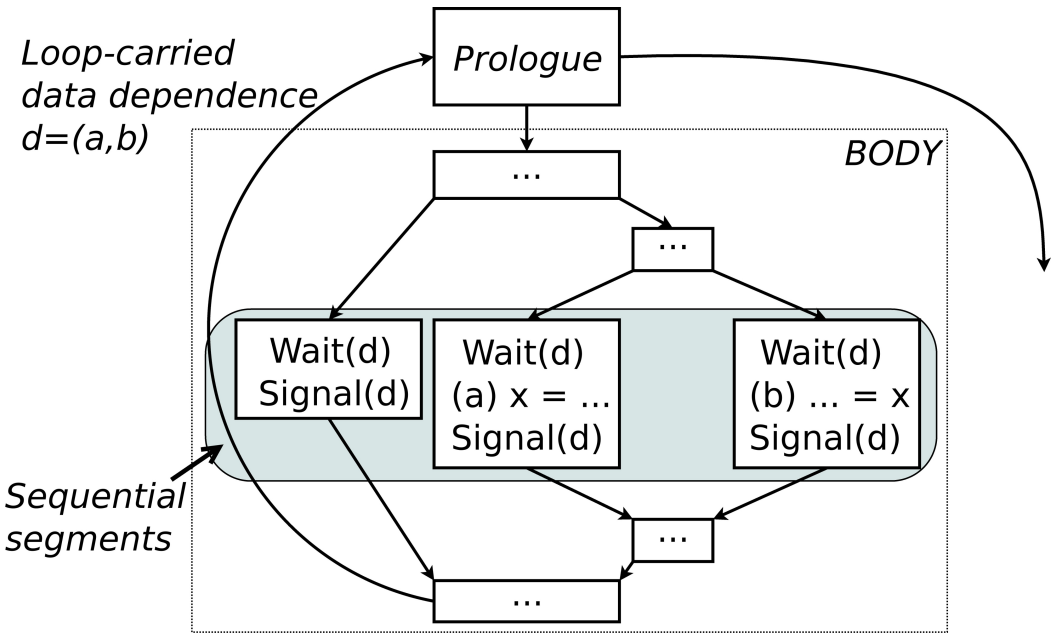
For every $d = (a, b) \in D_{Data}$:

- Instructions $Wait(d)$ are inserted as late as possible
- Instructions $Signal(d)$ are inserted as early as possible

Step 4: Computing Sequential Segments

For every $d = (a, b) \in D_{Data}$:

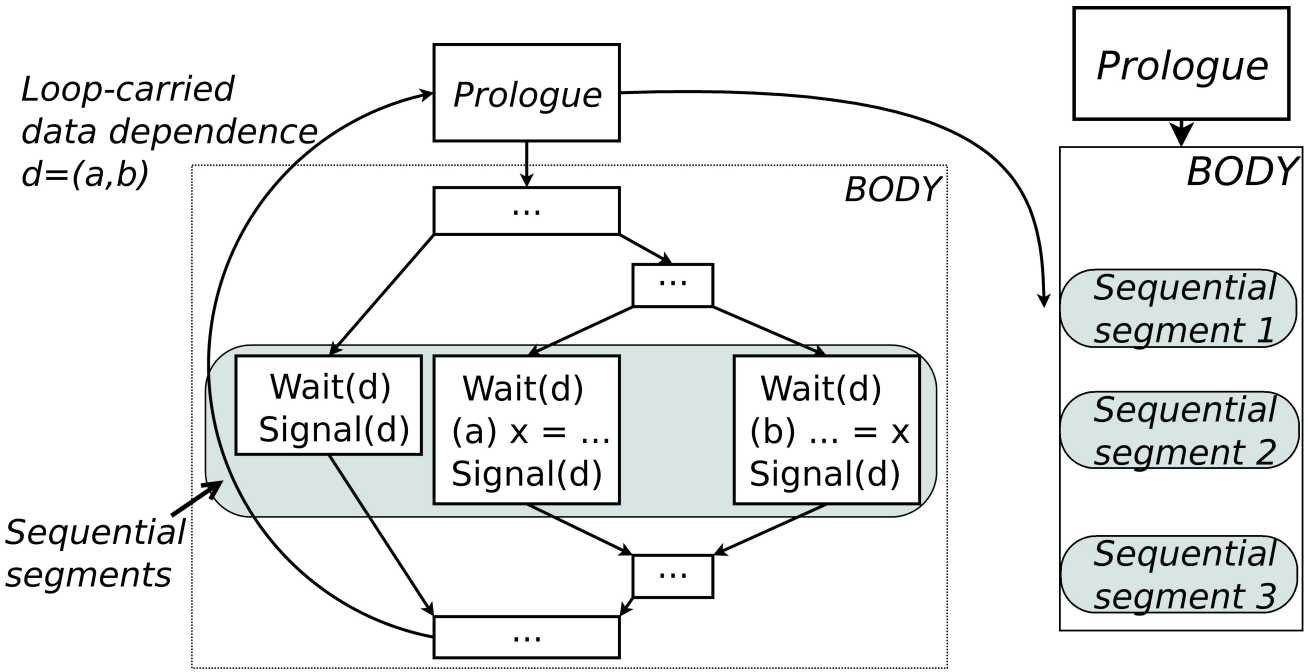
- Instructions $Wait(d)$ are inserted as late as possible
- Instructions $Signal(d)$ are inserted as early as possible



Step 4: Computing Sequential Segments

For every $d = (a, b) \in D_{Data}$:

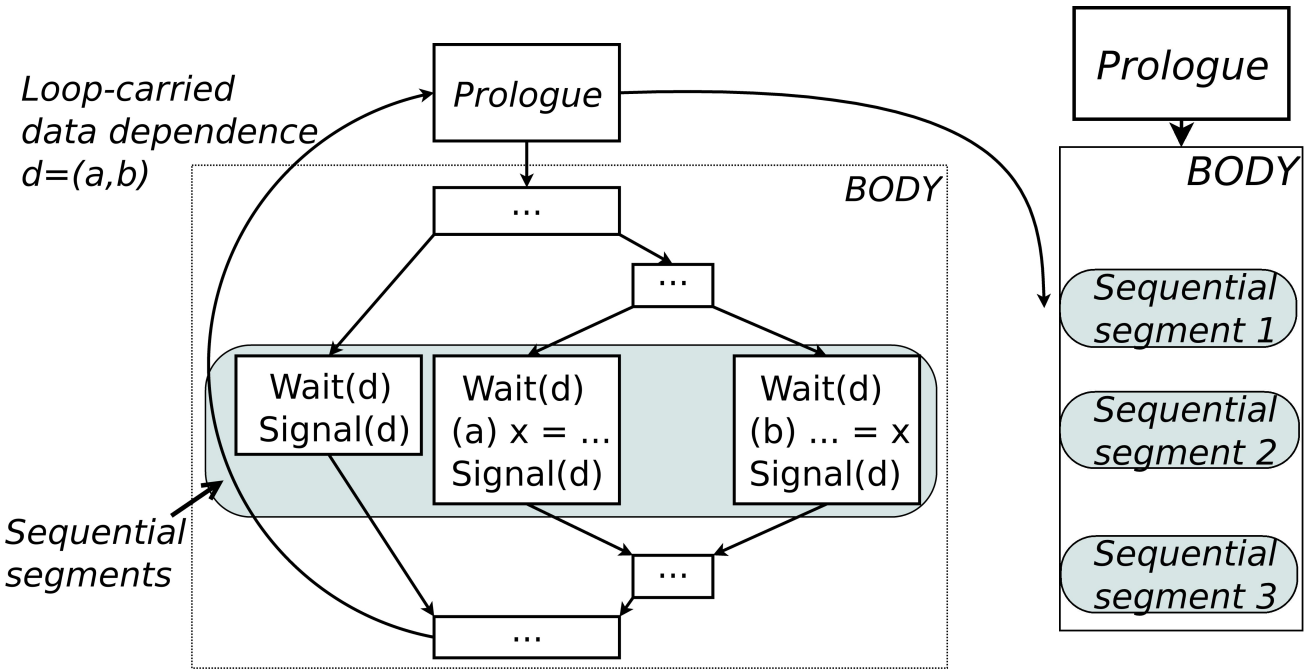
- Instructions $Wait(d)$ are inserted as late as possible
- Instructions $Signal(d)$ are inserted as early as possible

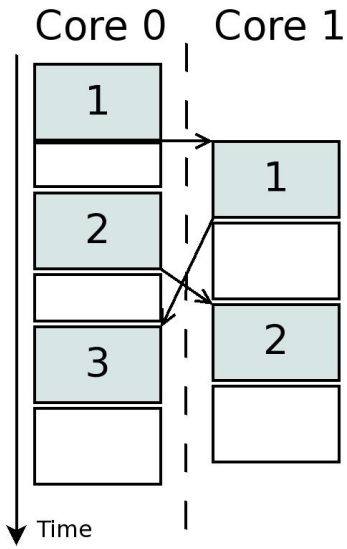


Step 4: Computing Sequential Segments

For every $d = (a, b) \in D_{Data}$:

- Instructions $Wait(d)$ are inserted as late as possible
- Instructions $Signal(d)$ are inserted as early as possible

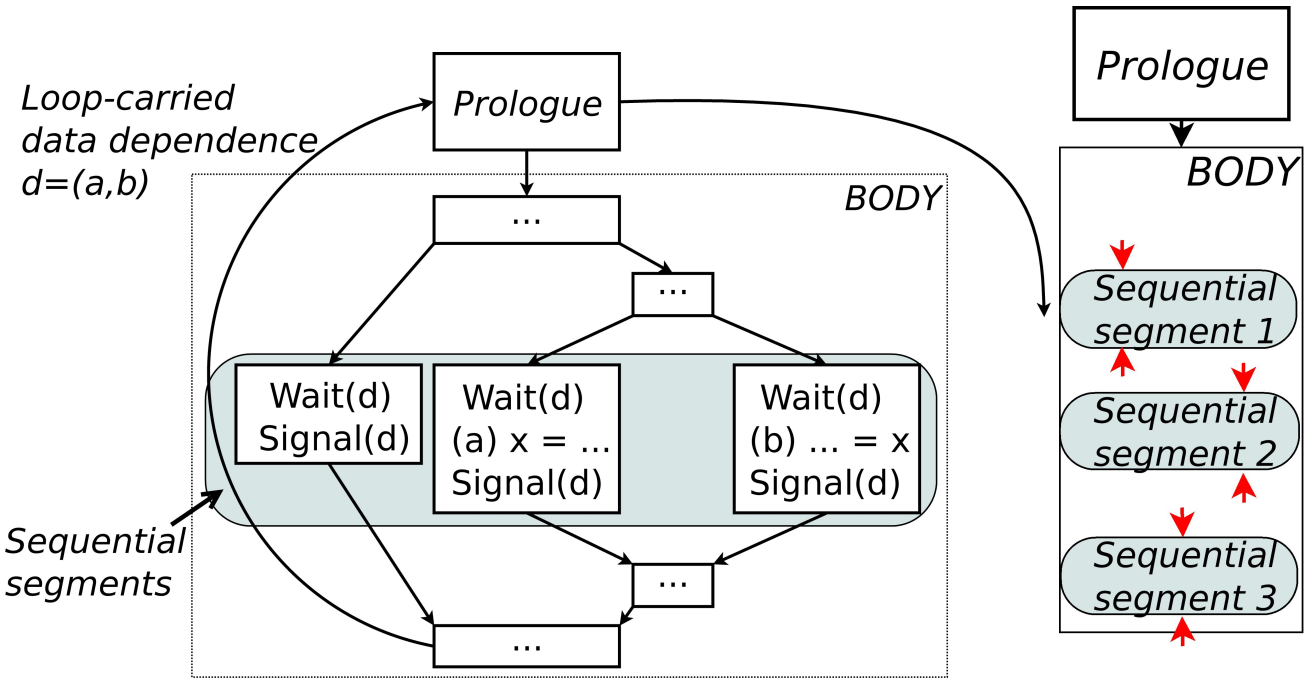


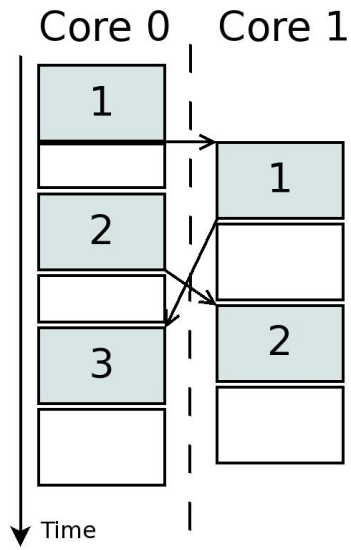


TLP among segments

Step 5: Minimizing Sequential Segments

Method inlining and code scheduling applied





TLP among segments

Steps 6 and 7

Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

Steps 6 and 7

Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

- intra- and inter-data dependences

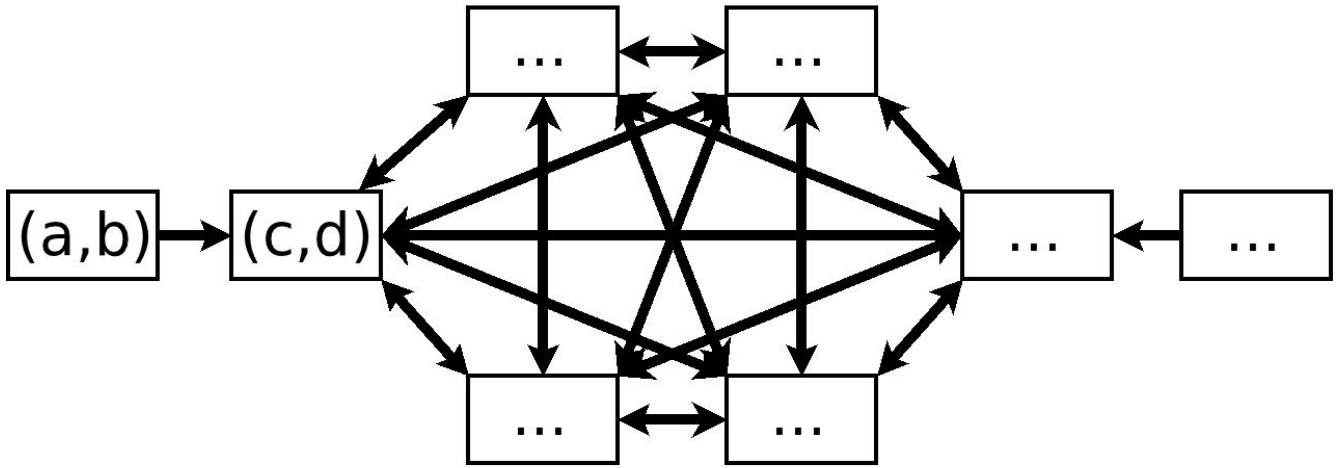
Theorem

Let $G = (N, E)$ be a data dependence redundance graph and let $N_{to\text{-}synch} \subseteq N$ be the set of dependences that includes every node without incoming edges and one node per cycle of G. Synchronizing the set $N_{to\text{-}synch}$ synchronizes the entire set of dependences N.

Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

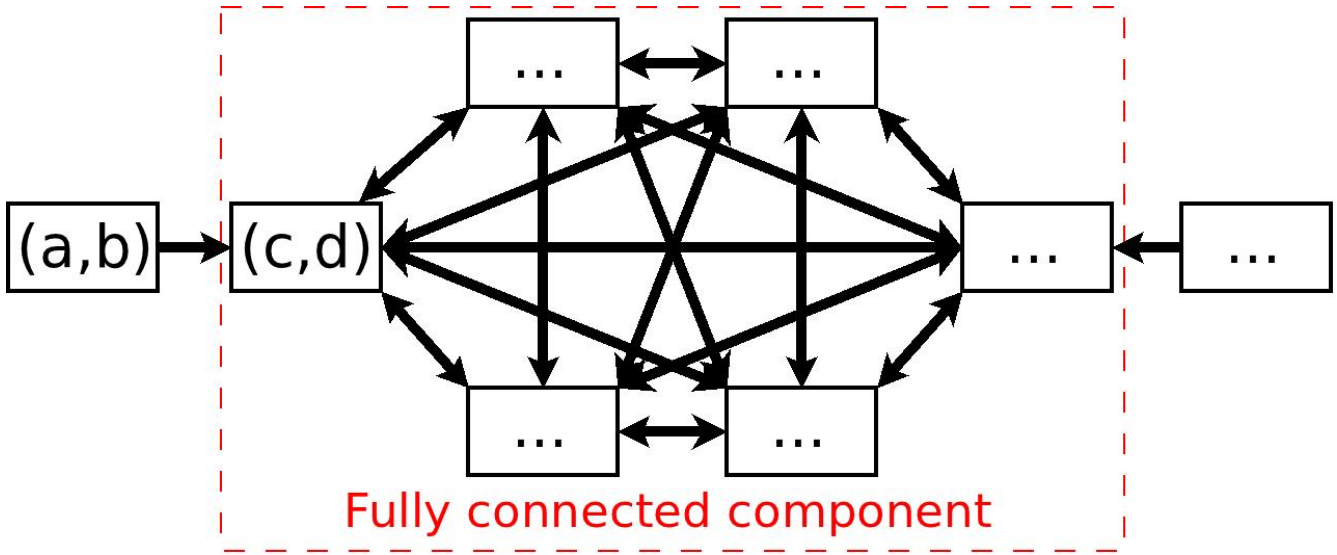
- intra- and inter-data dependences



Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

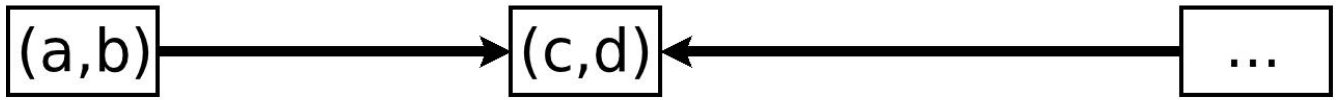
- intra- and inter-data dependences



Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

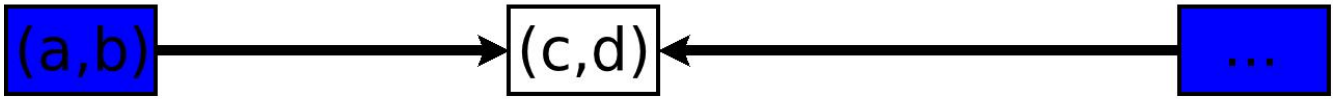
- intra- and inter-data dependences



Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

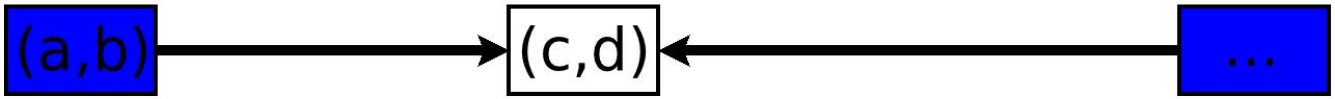
- intra- and inter-data dependences



Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

- intra- and inter-data dependences
- 80% – 98% of signals sent removed

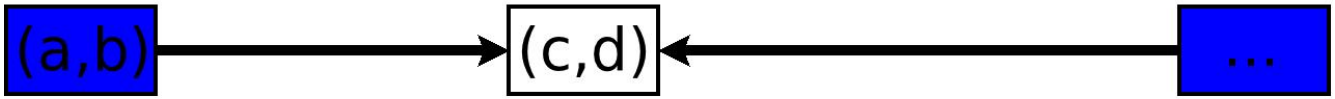


Steps 6 and 7

Step 6: Minimizing Signals

New analysis developed to minimize redundancy of signals

- intra- and inter-data dependences
- 80% – 98% of signals sent removed

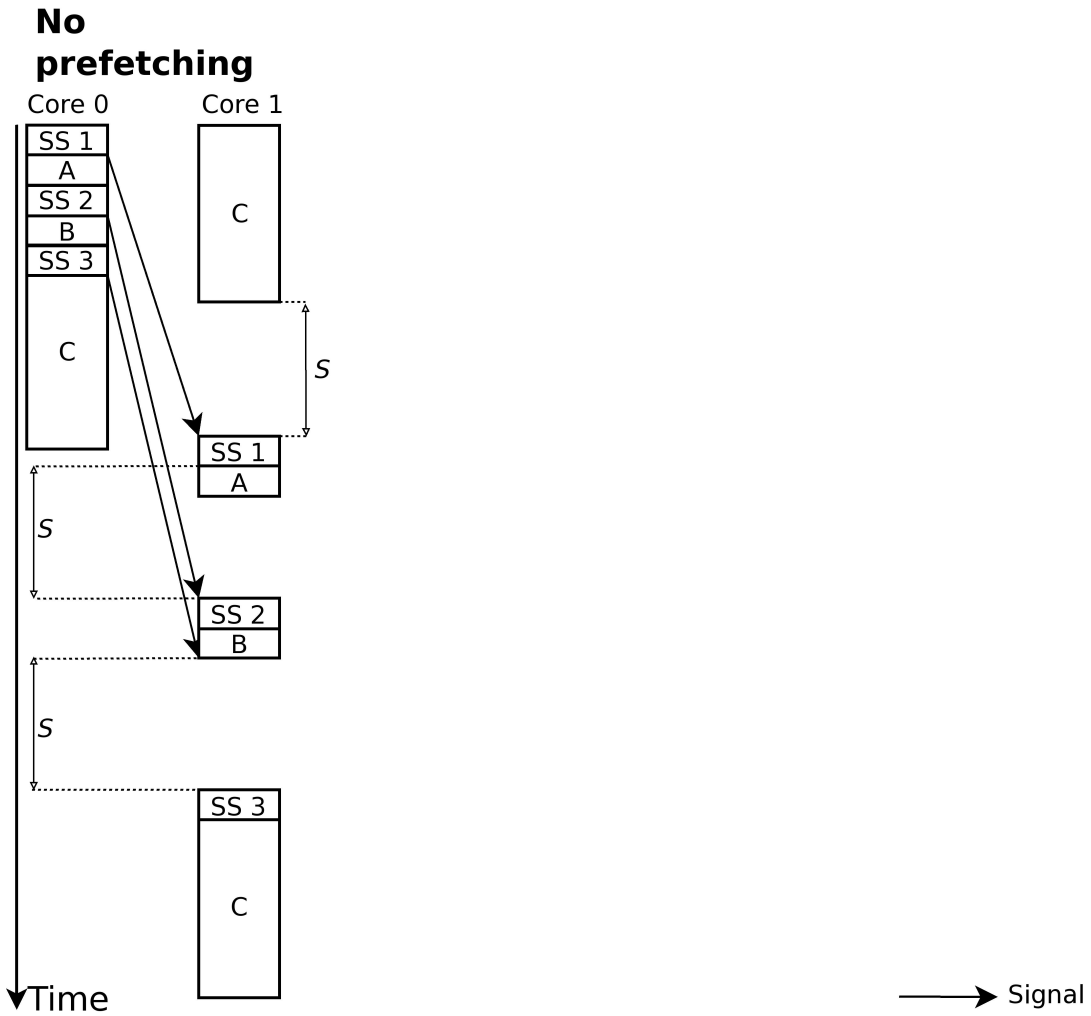


Step 7: Inserting Inter-Thread Communication

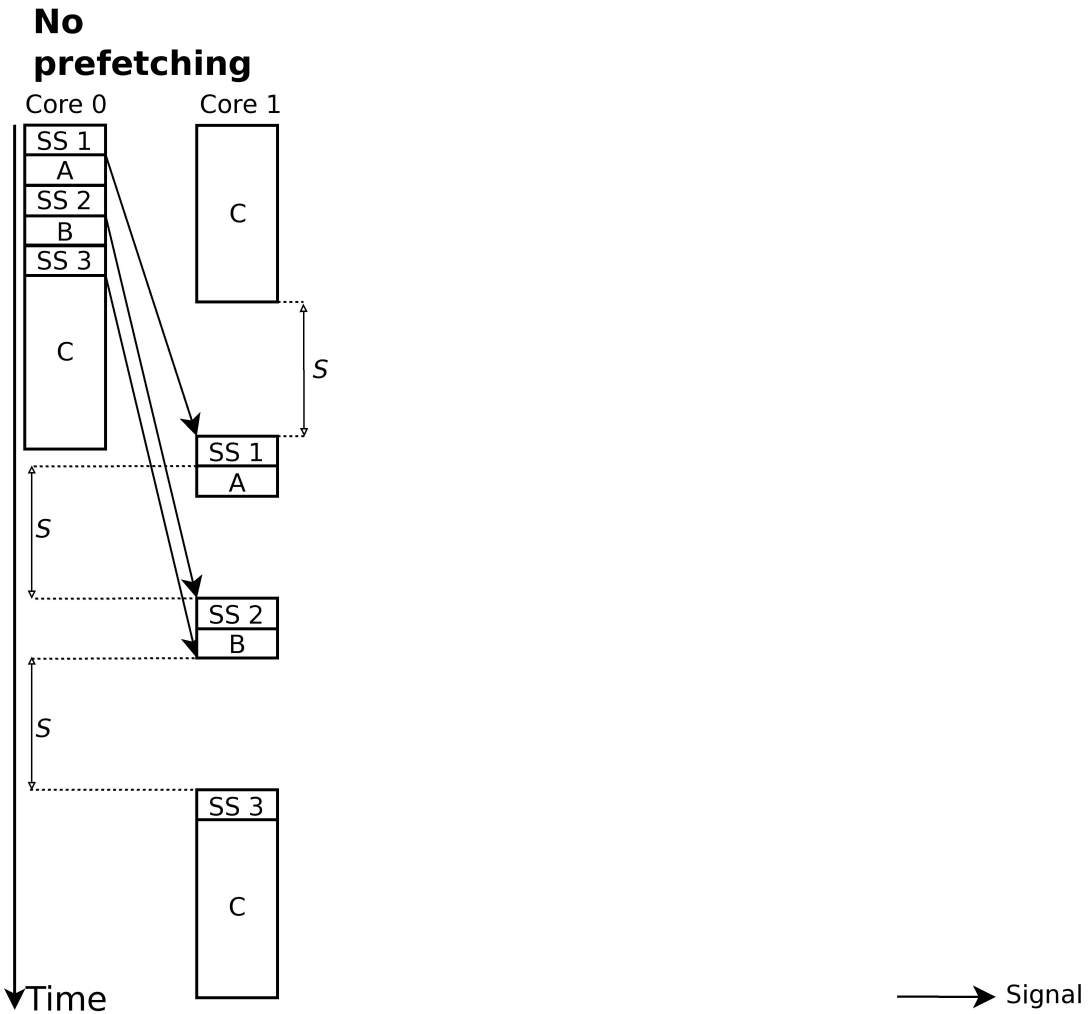
New analysis to minimize loads and stores of shared locations

Step 8: Coupling with Helper Threads

Step 8: Coupling with Helper Threads

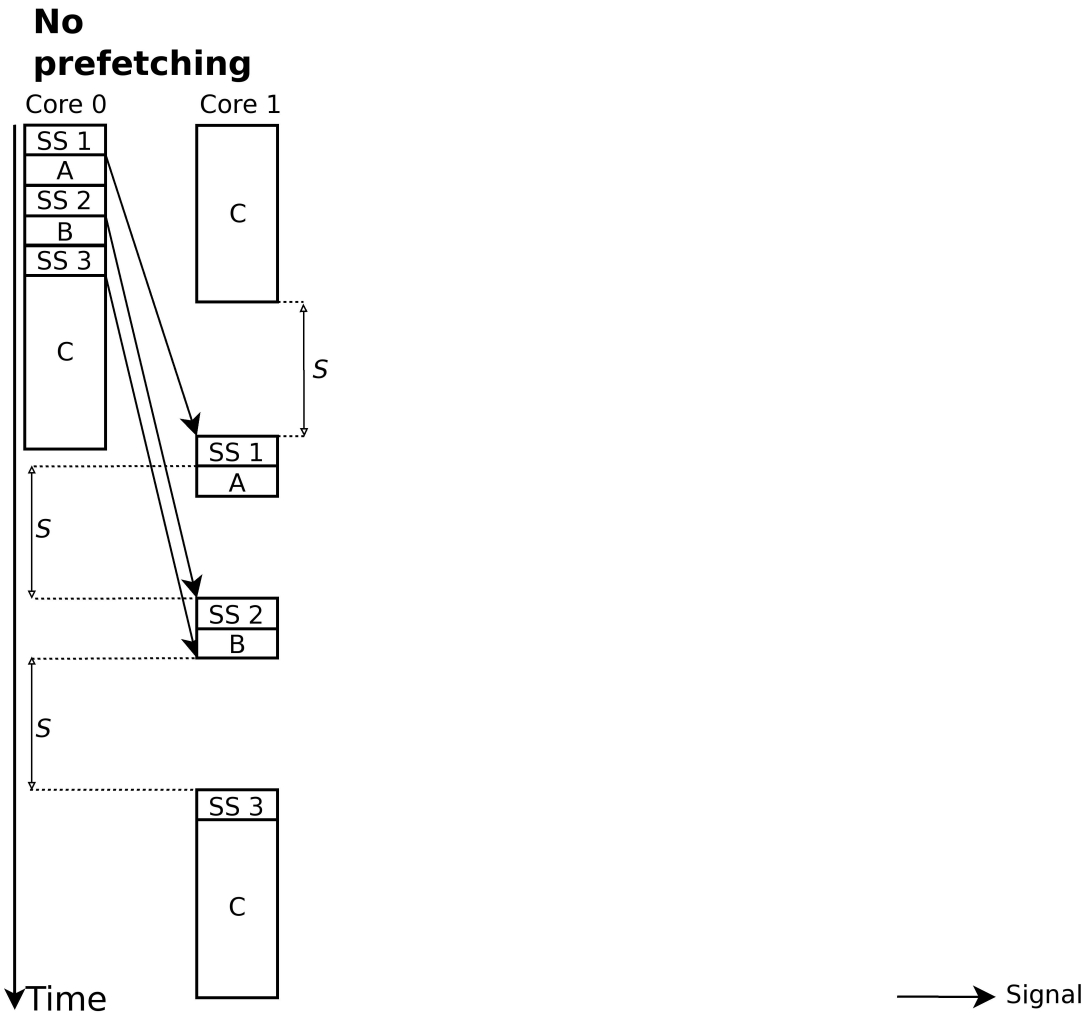


Step 8: Coupling with Helper Threads



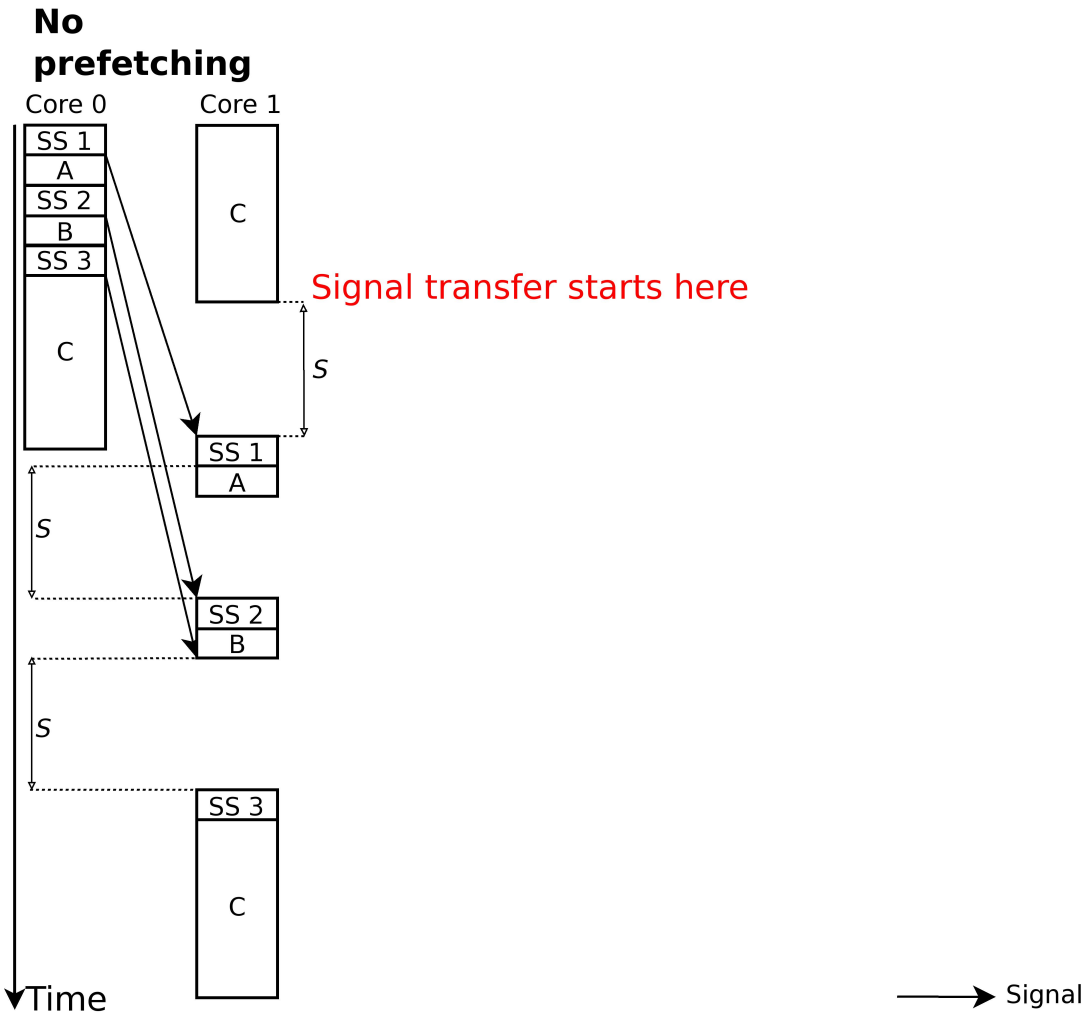
- Cache memories are pull systems

Step 8: Coupling with Helper Threads



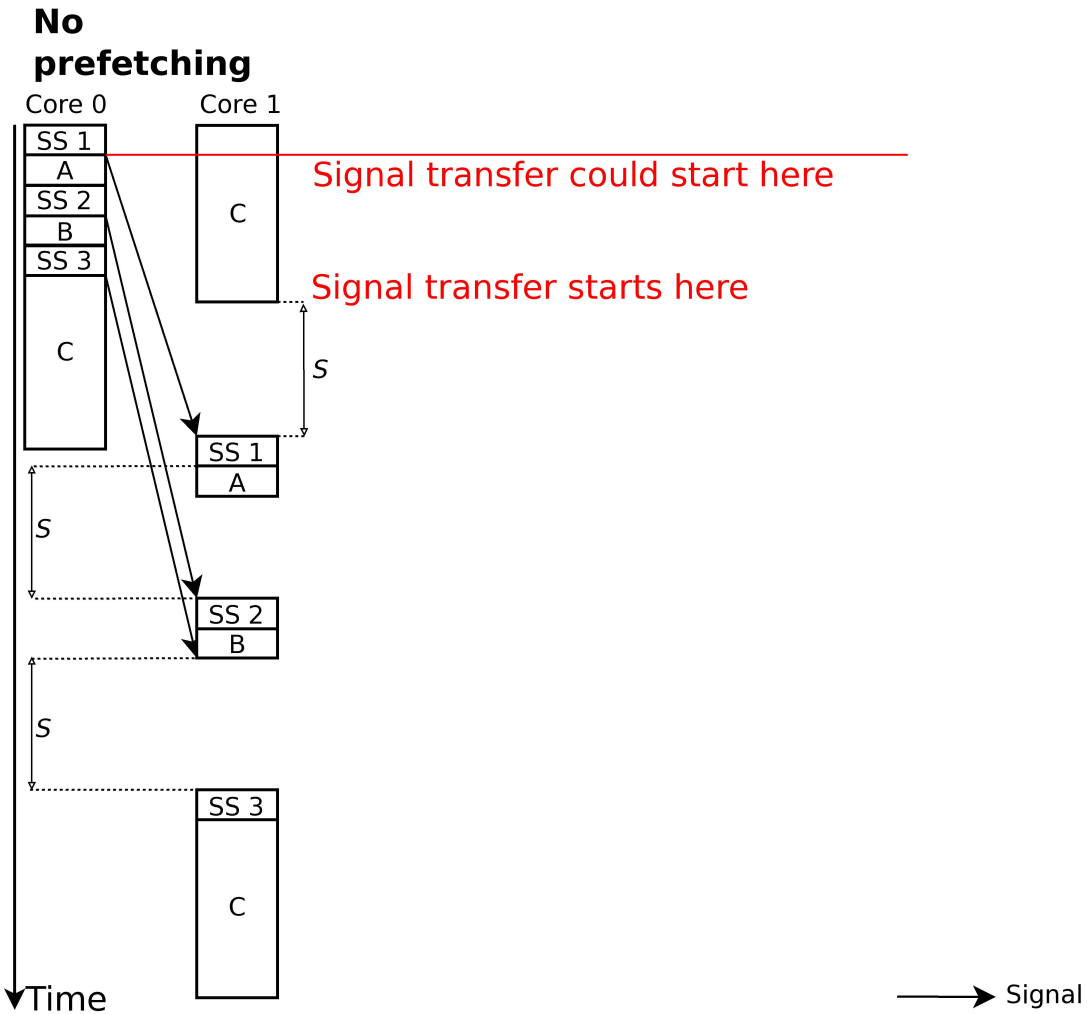
- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching

Step 8: Coupling with Helper Threads



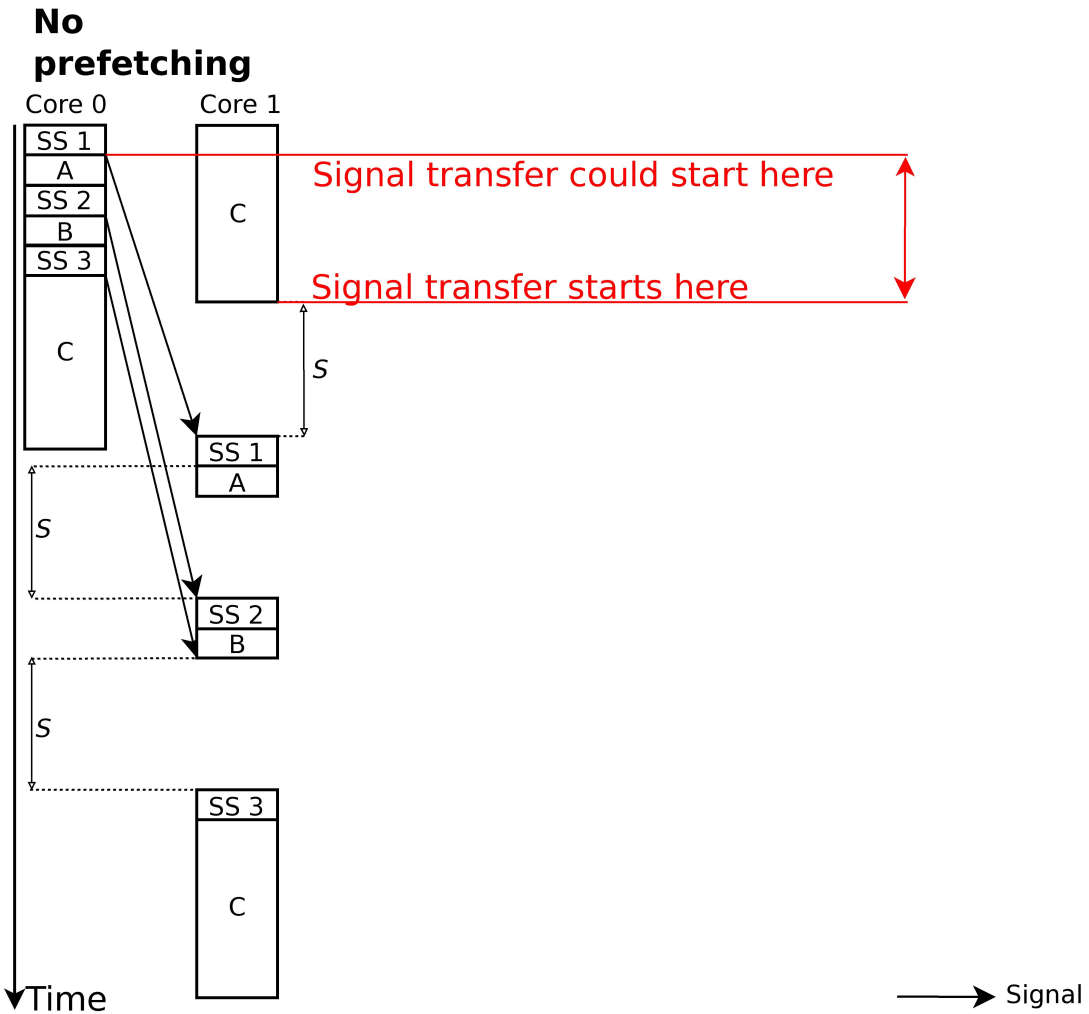
- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching

Step 8: Coupling with Helper Threads



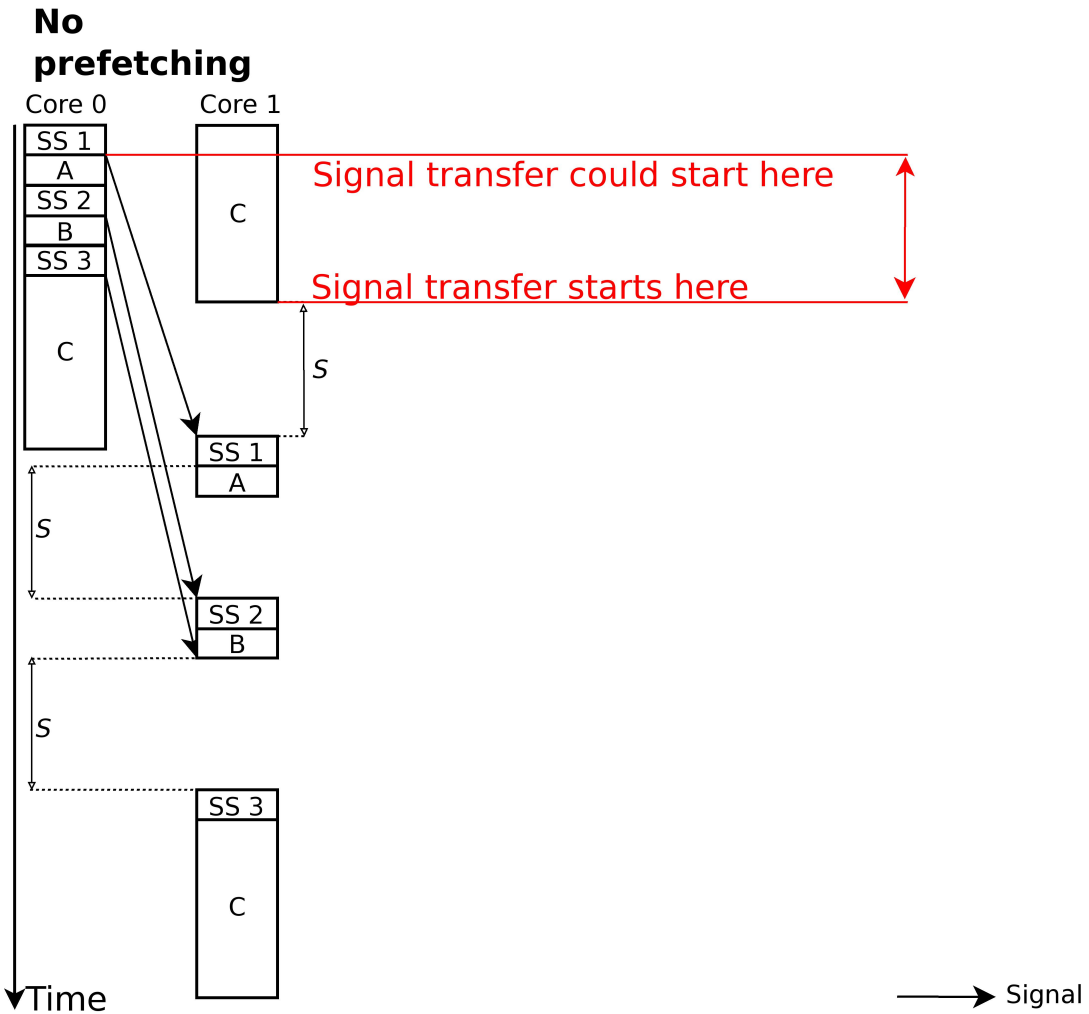
- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching

Step 8: Coupling with Helper Threads



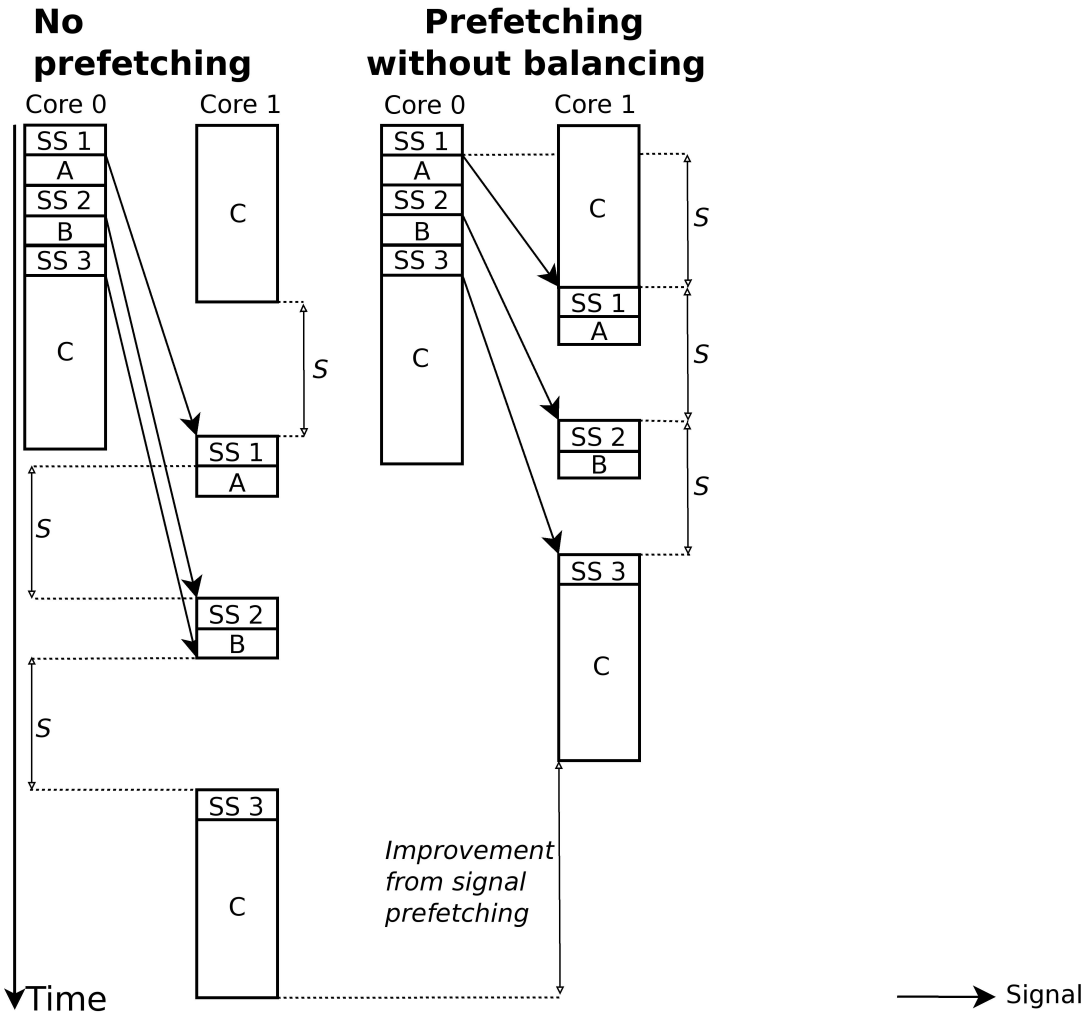
- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching

Step 8: Coupling with Helper Threads



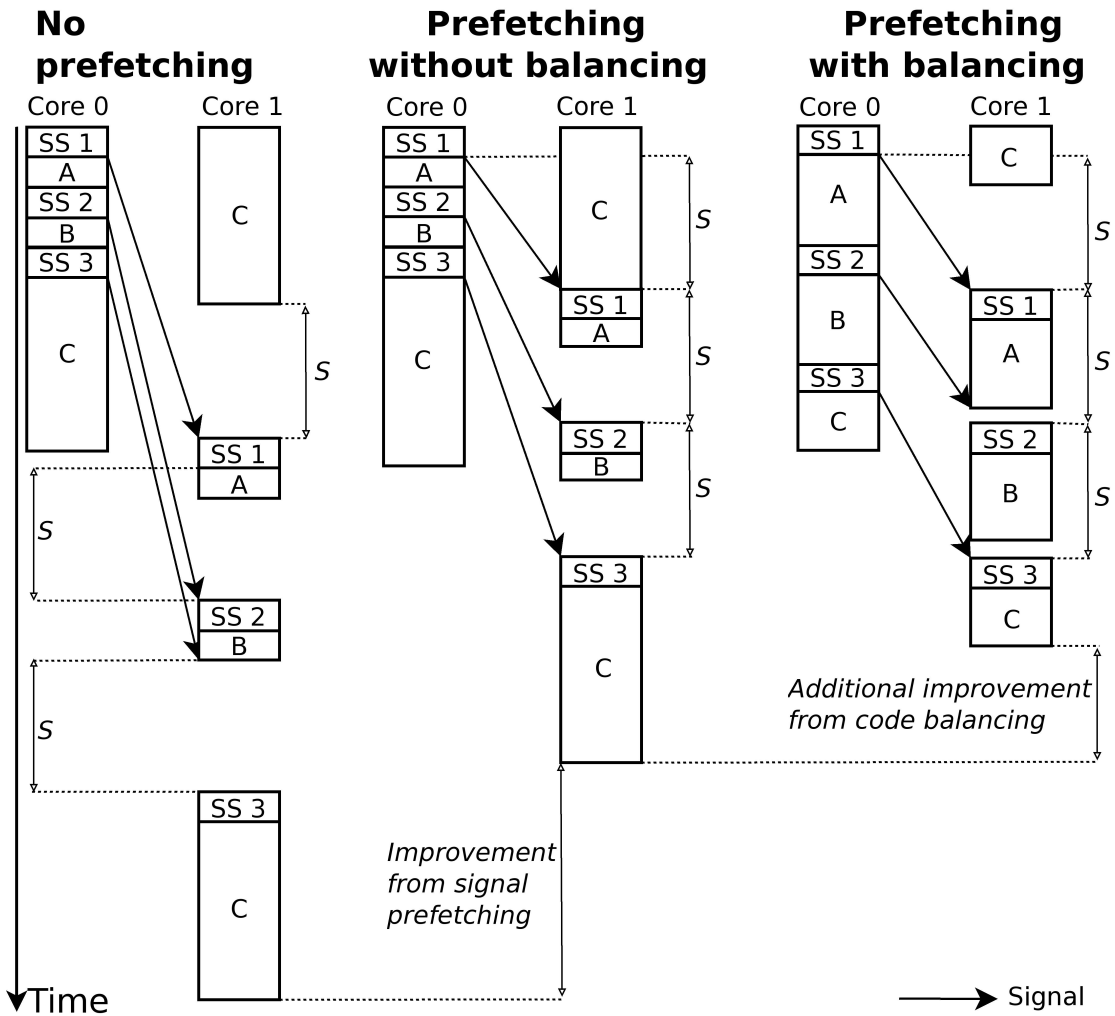
- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching
- Observation: sequence of sequential segments **predictable**

Step 8: Coupling with Helper Threads



- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching
- Observation: sequence of sequential segments **predictable**

Step 8: Coupling with Helper Threads



- Cache memories are pull systems
- Solution: couple helper threads for signal prefetching
- Observation: sequence of sequential segments **predictable**

Summary

- Motivation
- A simple idea
- Single loop parallelization
- Loop selection
- Evaluation
- Conclusion

Loop Selection

HELIX approach

Loop Selection

HELIX approach

- Each loop $\in$ program is analyzed independently

Loop Selection

HELIX approach

- Each loop $\in$ program is analyzed independently
- The program is analyzed to identify the most profitable loops

Single Loop Analysis

Assumption

Single Loop Analysis

Assumption

- Time spent to send a signal is
  - always $\in$ critical path

Single Loop Analysis

Assumption

- Time spent to send a signal is
  - always $\in$ critical path
  - constant

Single Loop Analysis

Assumption

- Time spent to send a signal is
  - always $\in$ critical path
  - constant

$$\text{Speedup} = \frac{Seq + Par}{Seq + \frac{Par}{N} + O}$$

Single Loop Analysis

Assumption

- Time spent to send a signal is
  - always $\in$ critical path
  - constant

$$\text{Speedup} = \frac{Seq + Par}{Seq + \frac{Par}{N} + O}$$

where

Overhead

$$O \approx Sig \times S + \left\lceil \frac{Bytes}{CPU_{word}} \right\rceil \times M$$

Single Loop Analysis

Assumption

- Time spent to send a signal is
  - always $\in$ critical path
  - constant

$$\text{Speedup} = \frac{Seq + Par}{Seq + \frac{Par}{N} + O}$$

where

Overhead

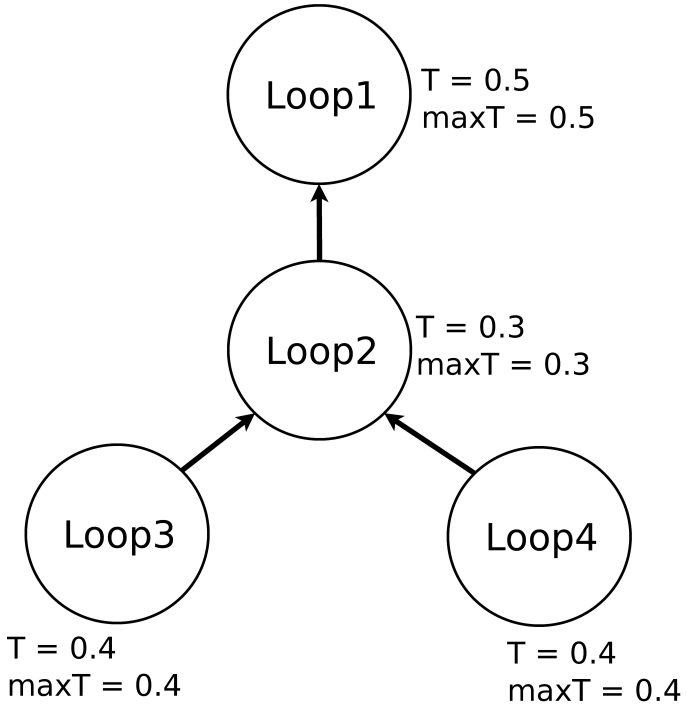
$$O \approx Sig \times S + \left\lceil \frac{Bytes}{CPU_{word}} \right\rceil \times M$$

Thanks to characteristic of the produced code:

$$Sig = |\text{loop iterations}| \times |\text{sequential segments}|$$

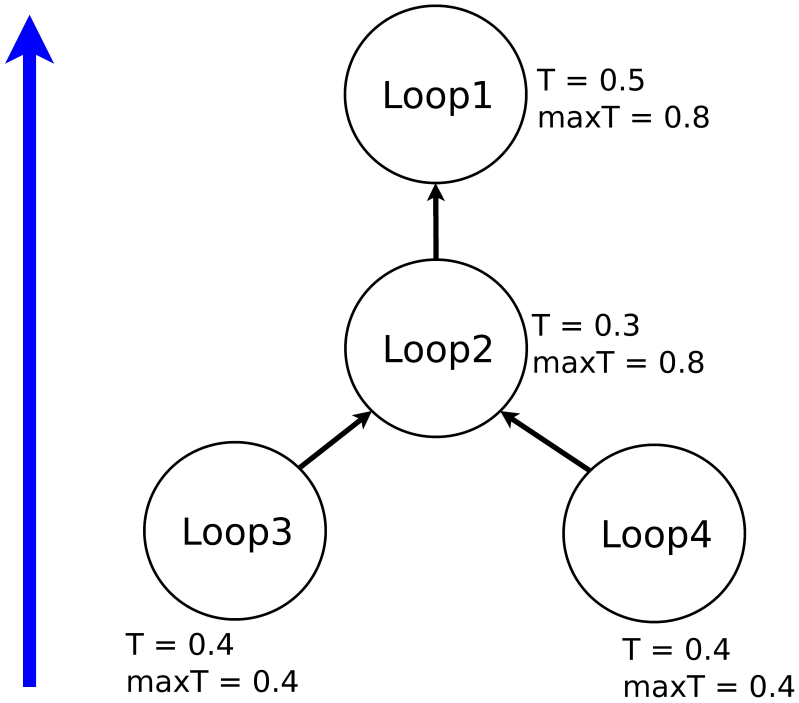
Identify loops to parallelize

Propagate parallel code information



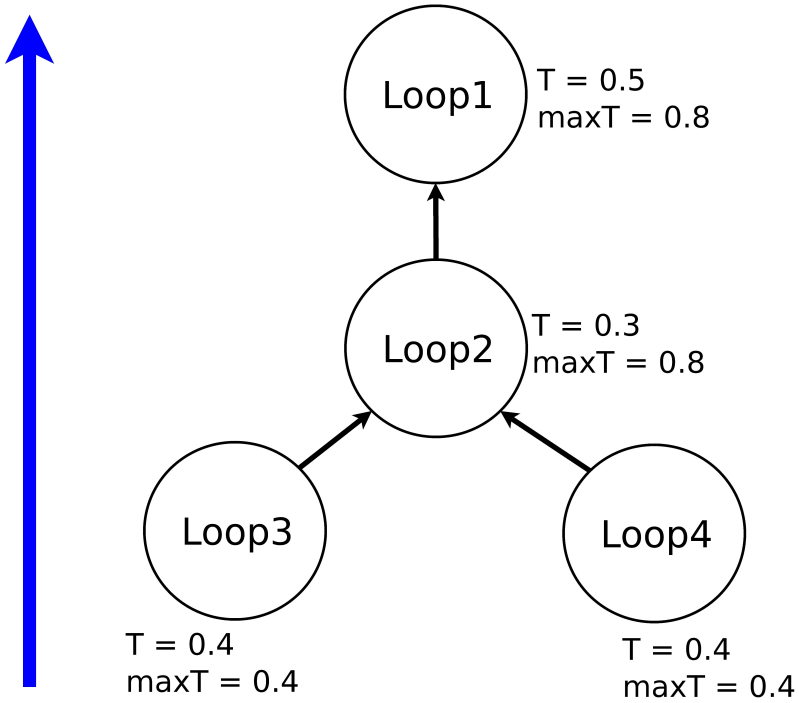
Identify loops to parallelize

Propagate parallel code information



Identify loops to parallelize

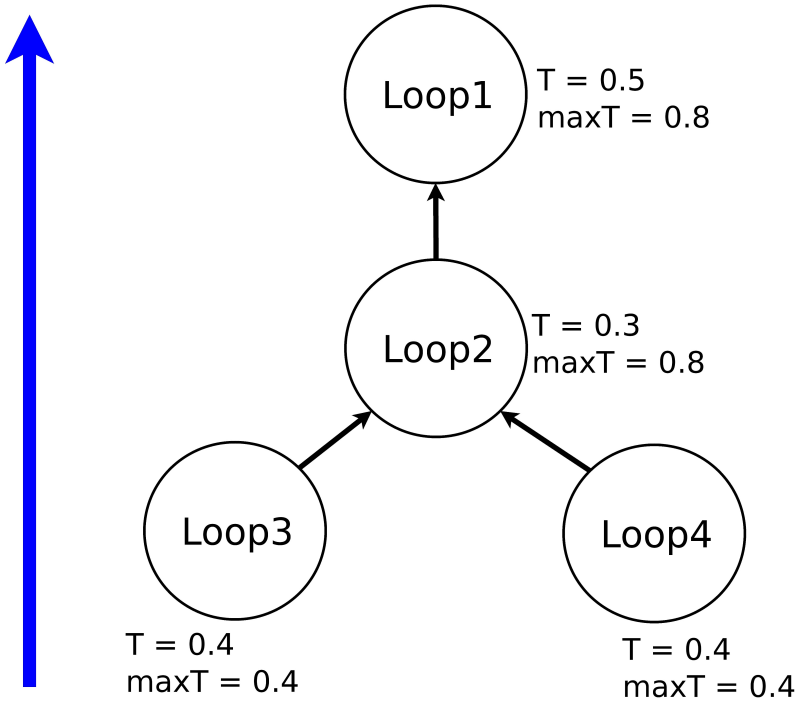
Propagate parallel code information



Notice: only max parallel is propagated

Identify loops to parallelize

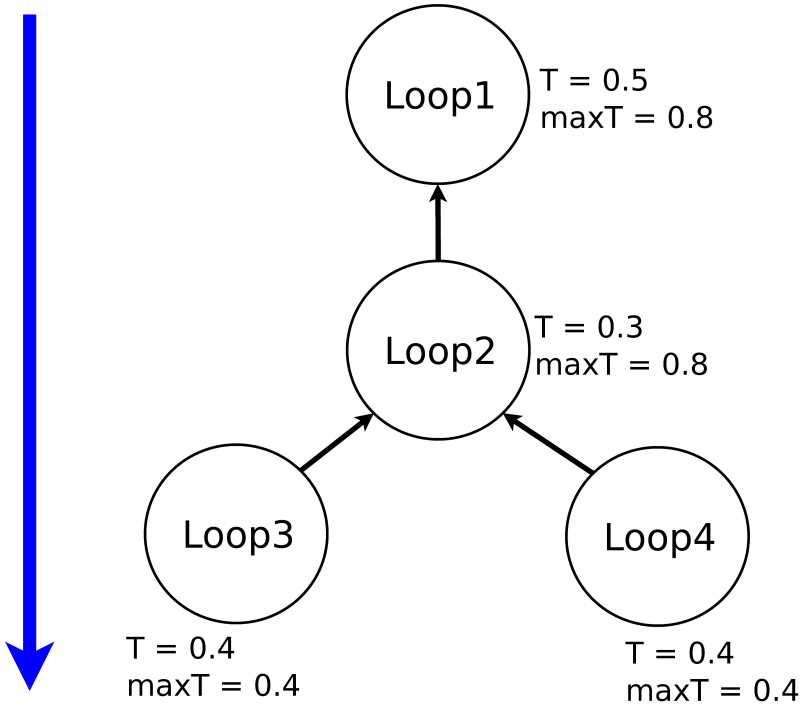
Exploit parallel code information



Notice: only max parallel is propagated

Identify loops to parallelize

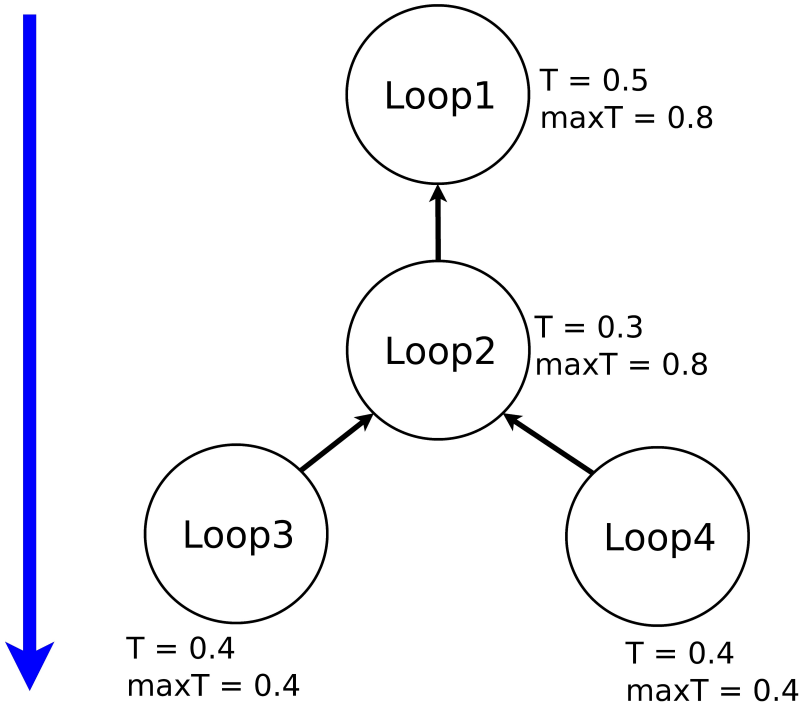
Exploit parallel code information



Notice: only max parallel is propagated

Identify loops to parallelize

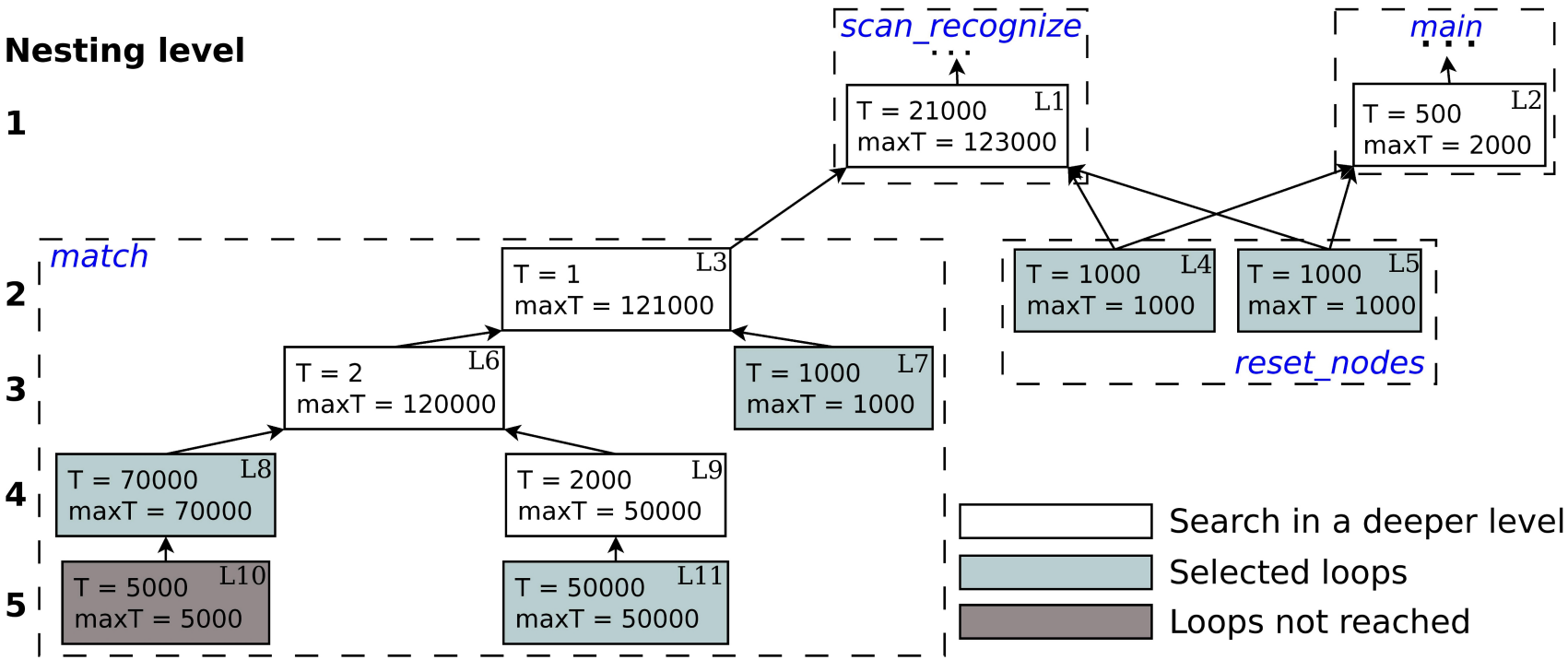
Exploit parallel code information



Notice: only max parallel is propagated

Is this an heuristic?

Loop Selection for 179.art



Summary

- Motivation
- A simple idea
- Single loop parallelization
- Loop selection
- Evaluation
- Conclusion

Evaluation

Platform

- Intel® Core™ i7-980X with six cores
  - Each operating at 3.33 GHz, with Turbo Boost disabled
- Three cache levels
  - The first two, 32KB and 256KB, are private to each core
  - All cores share the last level 12MB cache

Evaluation

Platform

- Intel® Core™ i7-980X with six cores
  - Each operating at 3.33 GHz, with Turbo Boost disabled
- Three cache levels
  - The first two, 32KB and 256KB, are private to each core
  - All cores share the last level 12MB cache

Benchmarks

C benchmarks from SPEC CPU2000

Evaluation

Platform

- Intel® Core™ i7-980X with six cores
  - Each operating at 3.33 GHz, with Turbo Boost disabled
- Three cache levels
  - The first two, 32KB and 256KB, are private to each core
  - All cores share the last level 12MB cache

Benchmarks

C benchmarks from SPEC CPU2000

Compiler

- HELIX has been implemented $\in$ static compiler ILDJIT
- C benchmarks are first translated to CIL bytecode by GCC4CLI

Evaluation

Platform

- Intel® Core™ i7-980X with six cores
  - Each operating at 3.33 GHz, with Turbo Boost disabled
- Three cache levels
  - The first two, 32KB and 256KB, are private to each core
  - All cores share the last level 12MB cache

Benchmarks

C benchmarks from SPEC CPU2000

Compiler

- HELIX has been implemented $\in$ static compiler ILDJIT
- C benchmarks are first translated to CIL bytecode by GCC4CLI

Evaluation

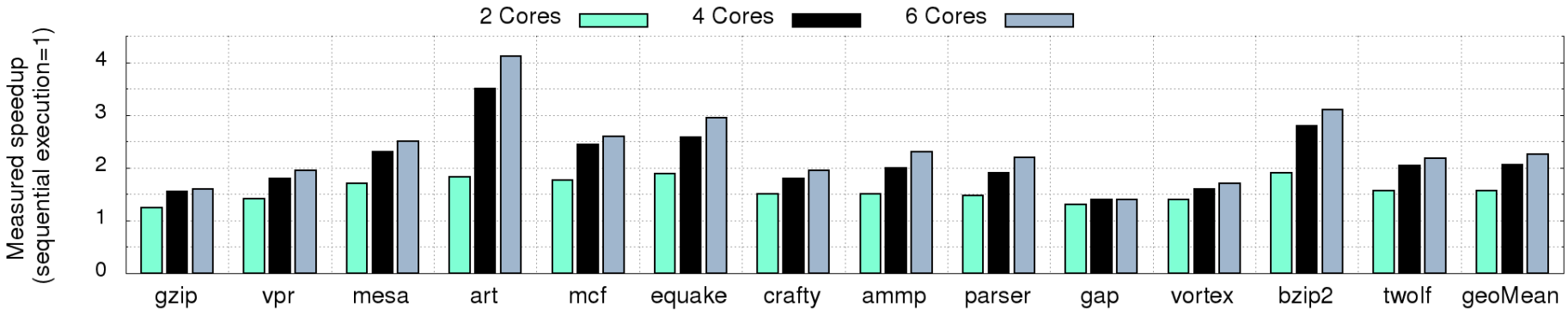
- The input *train* is used to select loops
- The input *ref* is used to compute the speedups

Speedup Obtained on a Real System

Overall program speedup

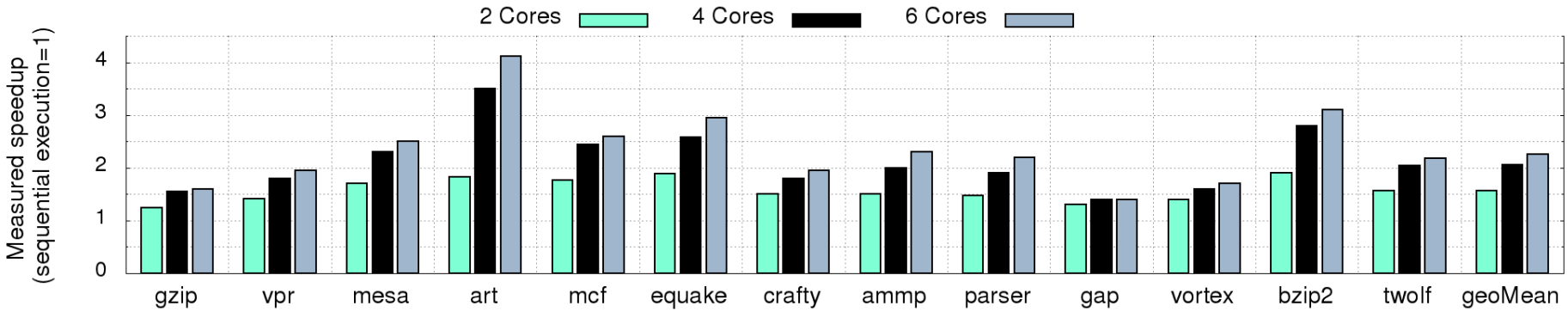
Speedup Obtained on a Real System

Overall program speedup



Speedup Obtained on a Real System

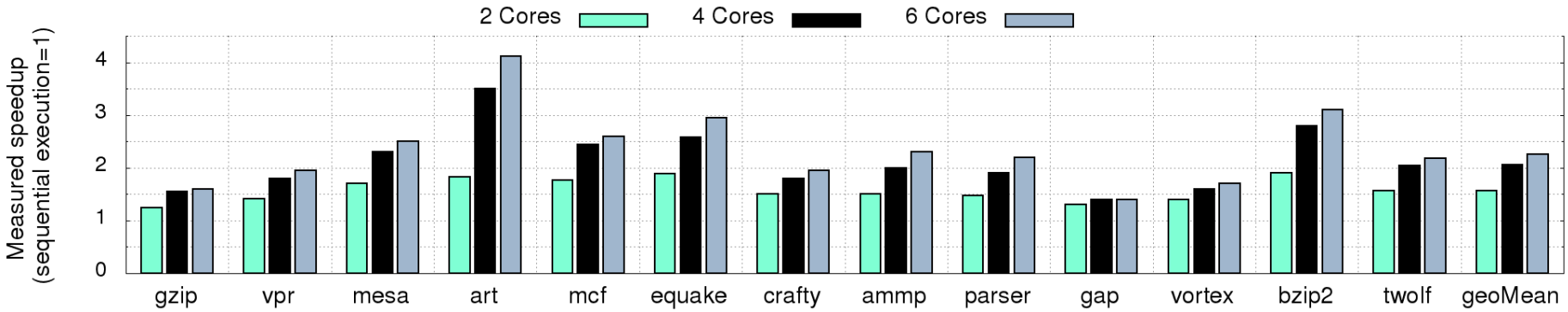
Overall program speedup



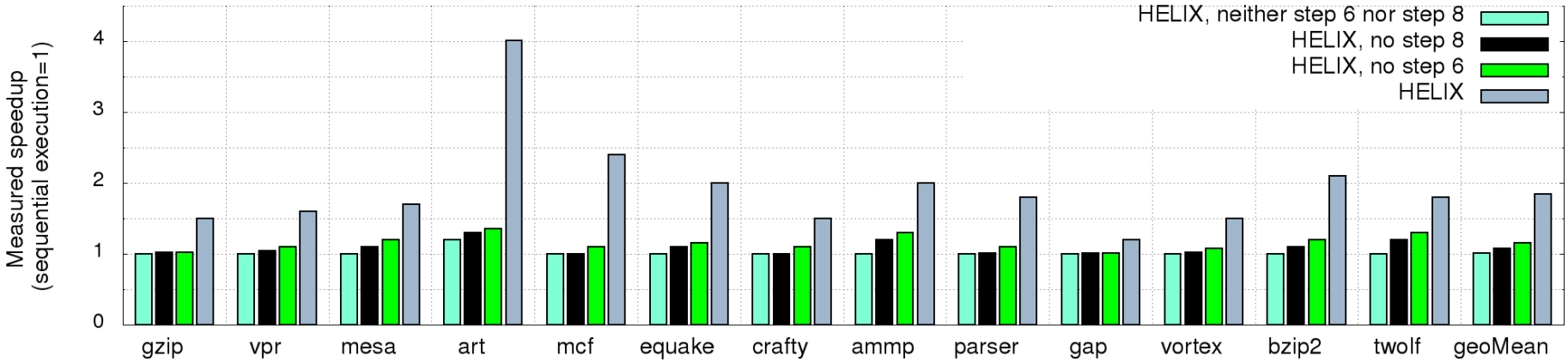
Most significant contributions

Speedup Obtained on a Real System

Overall program speedup

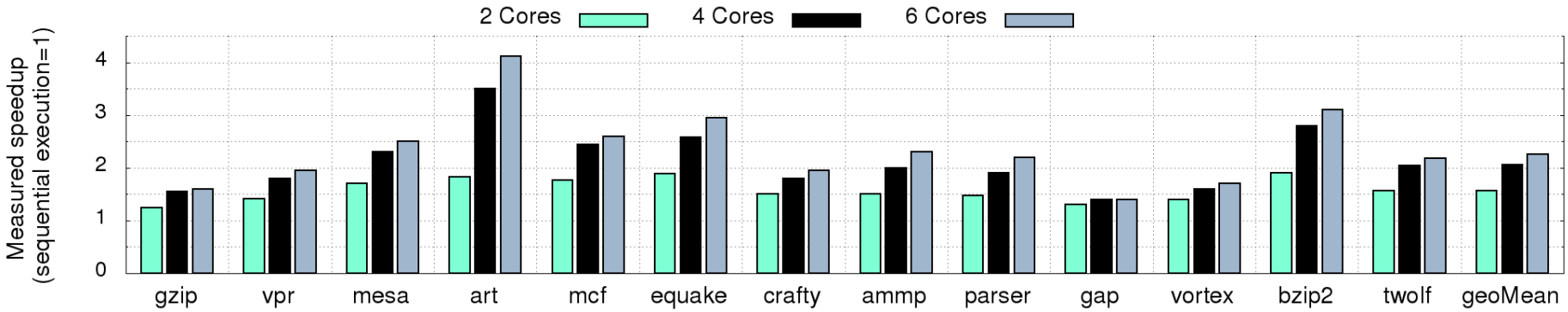


Most significant contributions

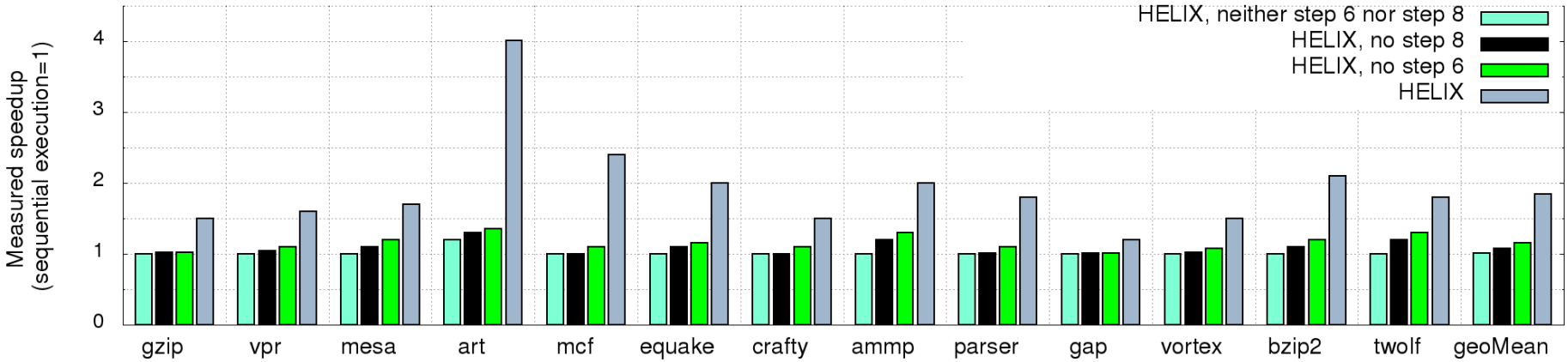


Speedup Obtained on a Real System

Overall program speedup



Most significant contributions



Notice: no slowdown

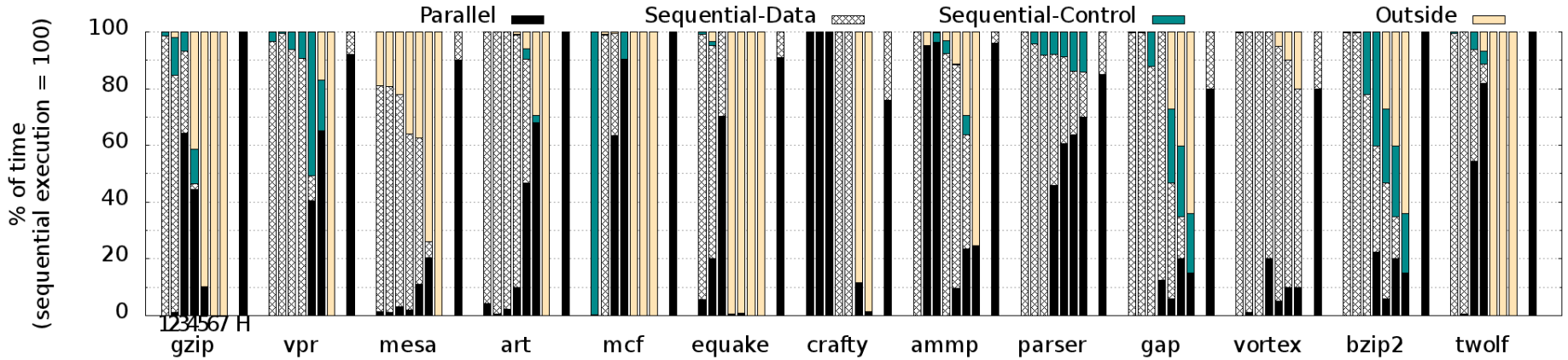
Chosen Loops

Chosen Loops

Most of the time is spent inside parallel code

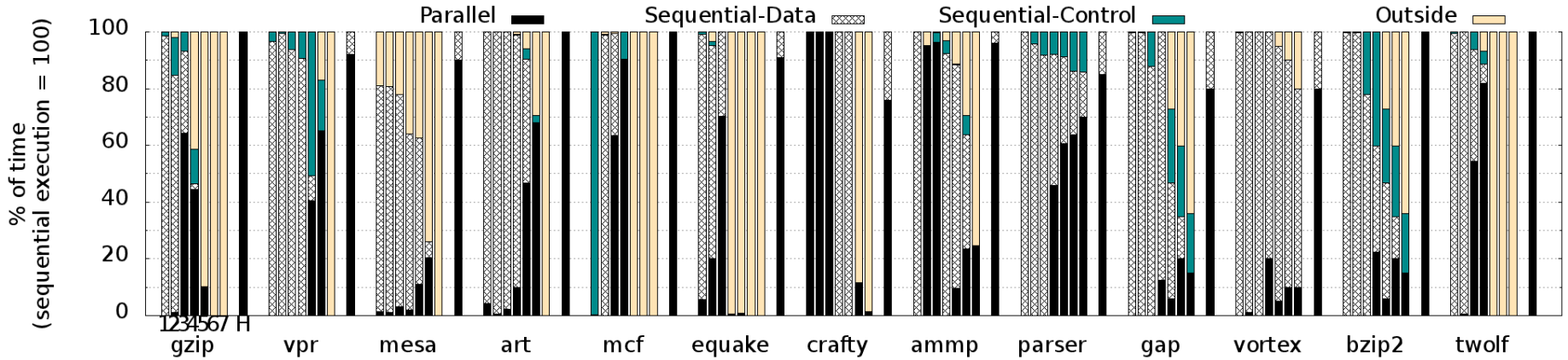
Chosen Loops

Most of the time is spent inside parallel code



Chosen Loops

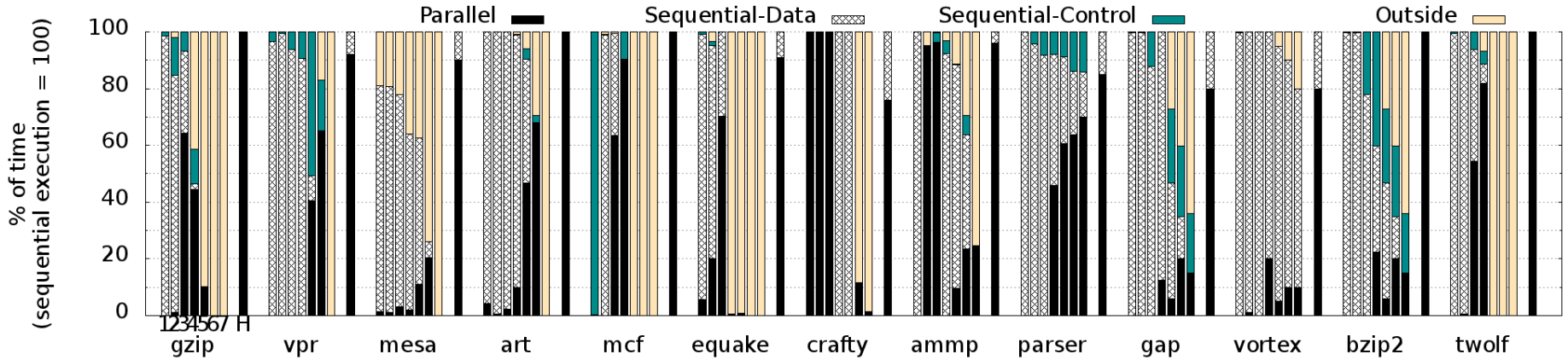
Most of the time is spent inside parallel code



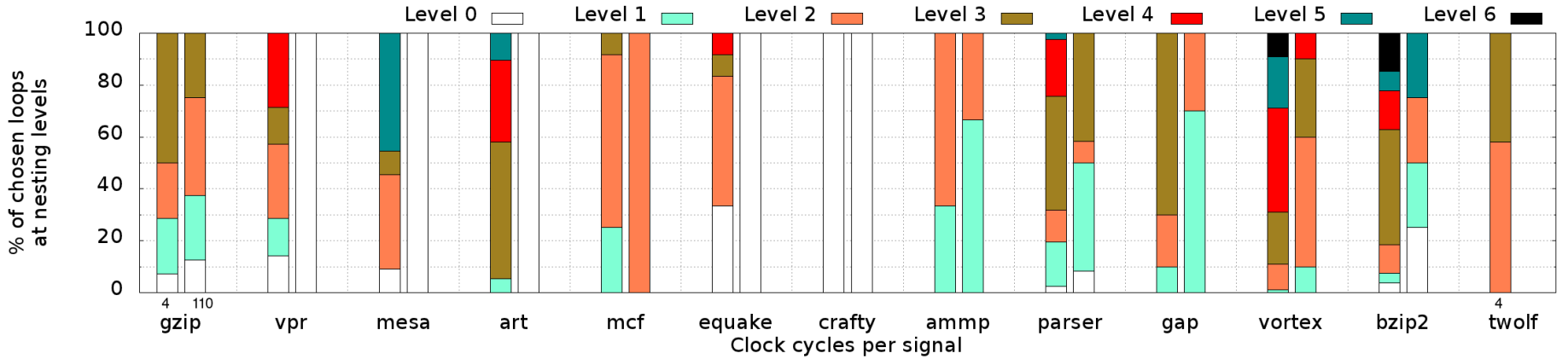
Loops $\in$ single nesting level is a poor solution

Chosen Loops

Most of the time is spent inside parallel code



Loops $\in$ single nesting level is a poor solution



Conclusion

HELIX: a new general purpose technique to extract parallelism

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to
  - Successfully identify the most profitable loops

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to
  - Successfully identify the most profitable loops
  - Avoid slowdowns

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to
  - Successfully identify the most profitable loops
  - Avoid slowdowns
  - Reduce delay per signal

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to
  - Successfully identify the most profitable loops
  - Avoid slowdowns
  - Reduce delay per signal
- How the hardware can be designed to improve HELIX code?

Conclusion

HELIX: a new general purpose technique to extract parallelism

- Significant speedups can be achieved on current hardware
  - Hardware not designed for this type of execution
- HELIX is able to run both independent and most of dependent code in parallel
- Thanks to the code predictability, HELIX is able to
  - Successfully identify the most profitable loops
  - Avoid slowdowns
  - Reduce delay per signal
- How the hardware can be designed to improve HELIX code?
- What are the limits of HELIX?

References

Websites

- HELIX
  - http://helix.eecs.harvard.edu
- ILDJIT
  - http://ildjit.sourceforge.net

Email

- xan@eecs.harvard.edu

Thanks for your attention!