Near-Optimal Instruction Selection on DAGs

David Ryan Koes

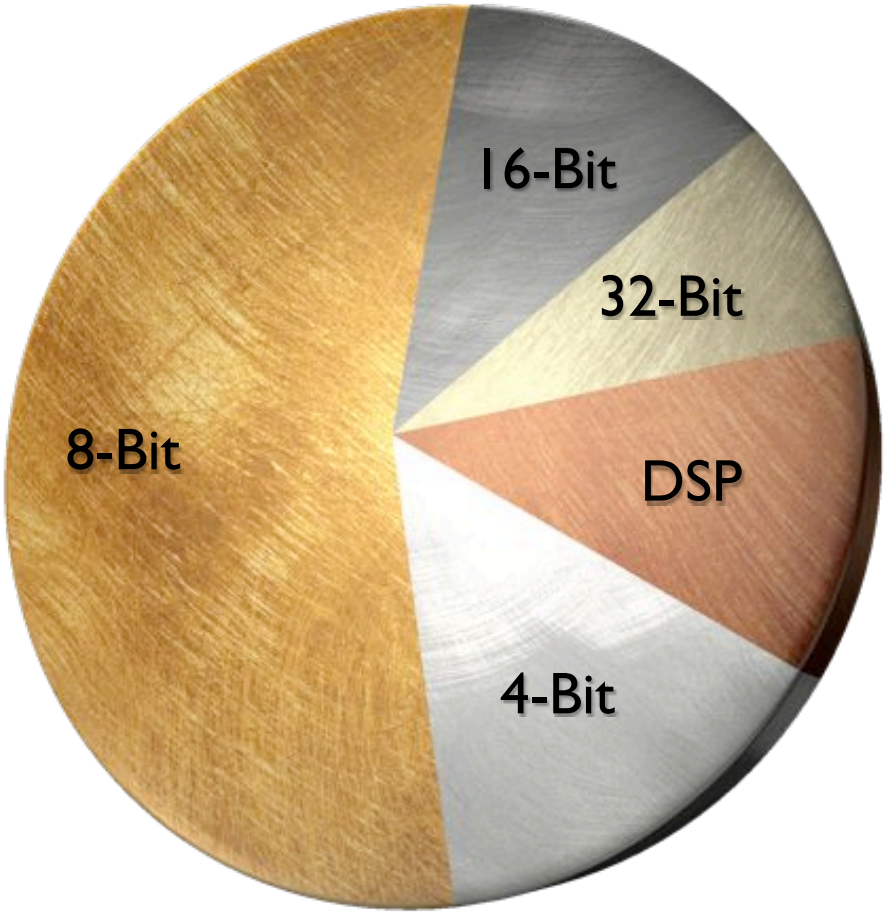
Seth Copen Goldstein

CGO 2008

4/7/2008

Embedded Processors by the Numbers

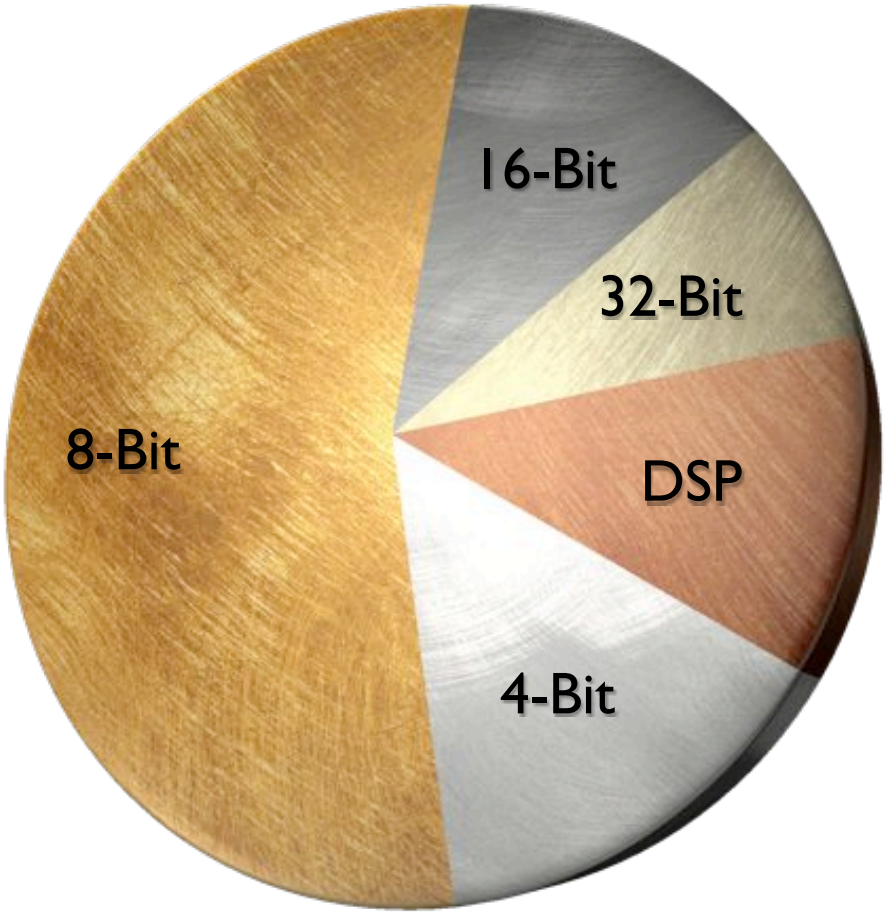
Microprocessors Sold by Type



*Most embedded processors are **resource constrained***

Embedded Processors by the Numbers

Microprocessors Sold by Type



*Most embedded processors are **resource constrained***

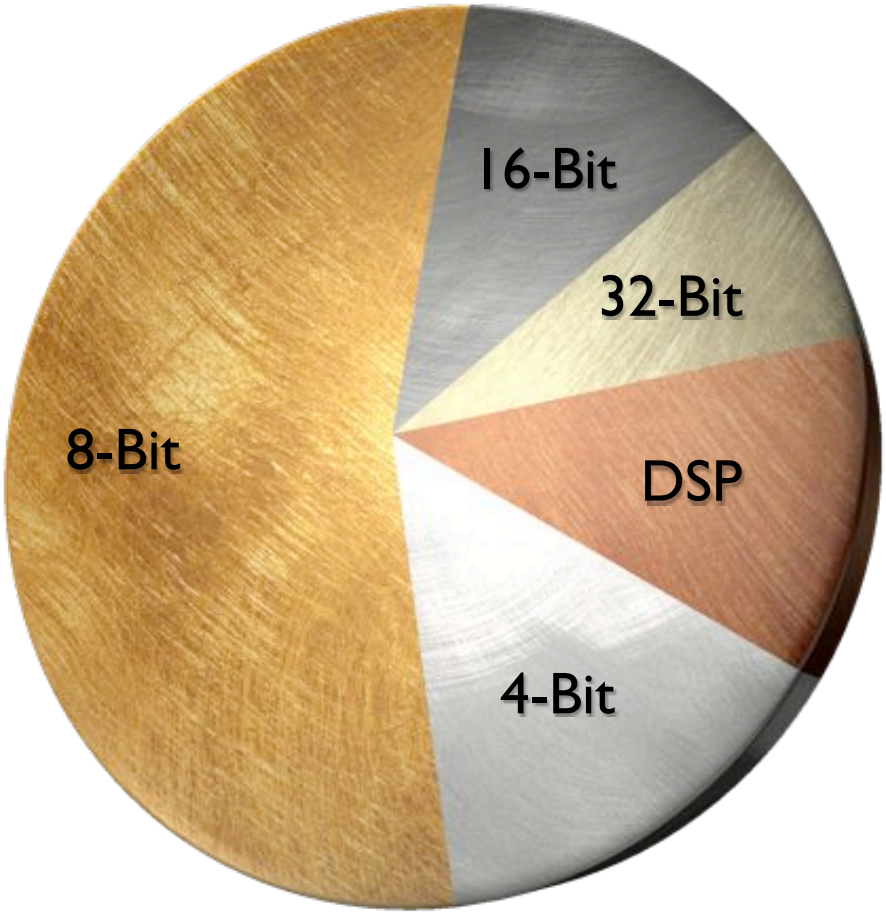
Resource Constraint: Memory

Example: Microchip PIC16F819

SRAM: 256 bytes
EEPROM: 256 bytes
Flash Memory: 3584 bytes

Embedded Processors by the Numbers

Microprocessors Sold by Type



*Most embedded processors are **resource constrained***

Resource Constraint: Memory

Example: Microchip PIC16F819

SRAM: 256 bytes

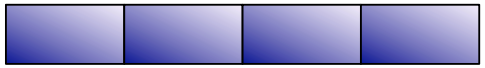
EEPROM: 256 bytes

Flash Memory: 3584 bytes

Limited instruction memory → code size critical

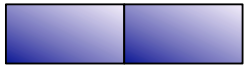
Architecting for Code Size

4 Byte Instructions



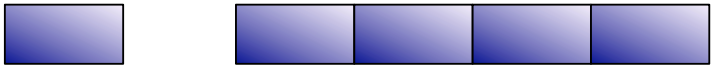
- Ample bits for accessing registers, supporting addressing modes, supporting ISA extensions
- Large code size

2 Byte Instructions



- Small code size, better instruction fetch
- Limited support for addressing modes, accessing registers; instruction count increases

Variable Sized Instructions



- Small code size; full support for addressing modes, accessing registers, ISA extensions
- Increases complexity of decoder, **compiler**

Complex Instruction Sets → Complex Compilers

Complex Instruction Sets

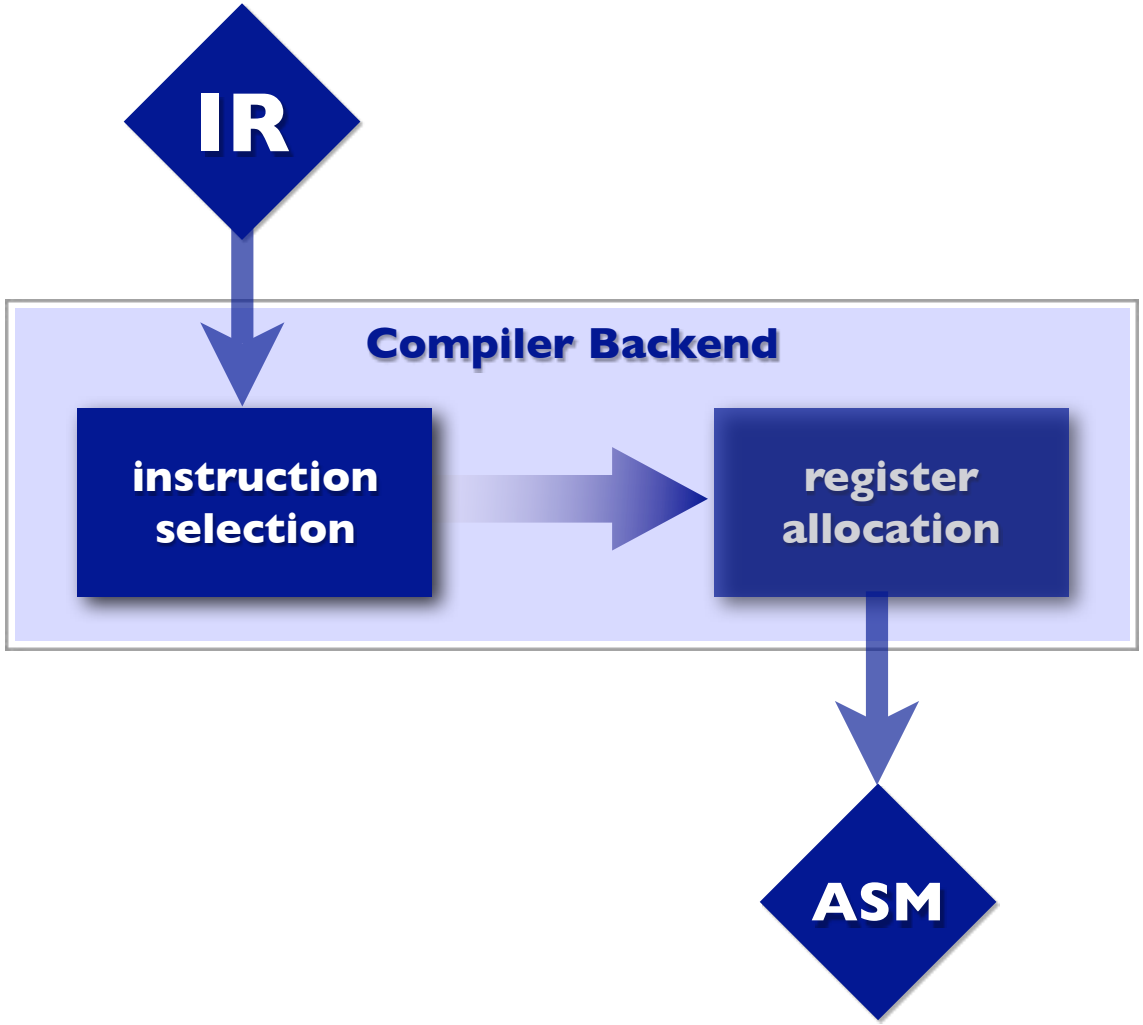
- variable length instructions
- full complement of addressing modes
- redundant instructions

x86 Example: **t+1**

`-incl t`	1 byte
`-addl $1,t`	3 bytes
`-leal 1(t),t`	3 bytes

The compiler must select the best instruction based upon its context

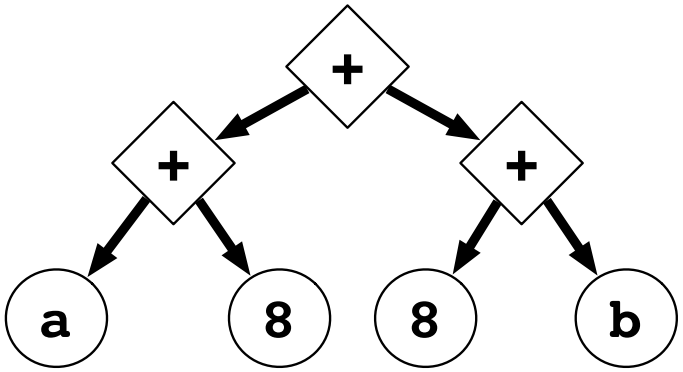
Instruction Selection



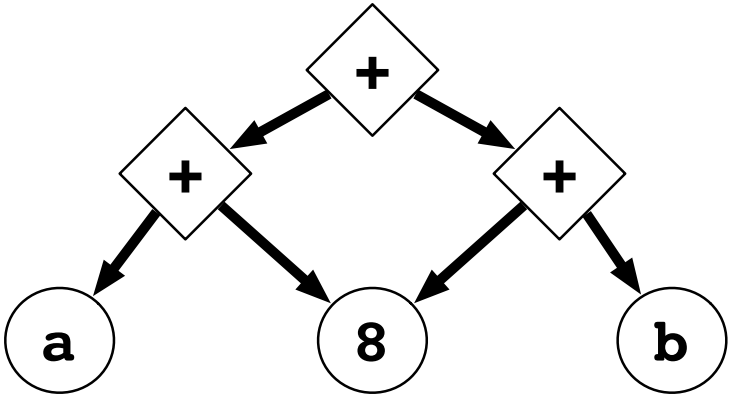
Intermediate Representations

```
(a+8) + (b+8);
```

Expression Tree



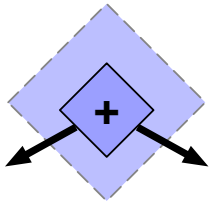
Expression DAG



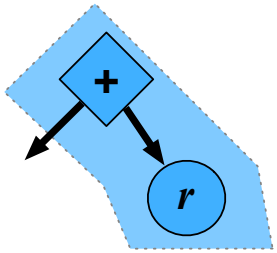
Explicitly encodes redundant computations

Linear IRs such as three address pseudo-assembly can be easily converted to a structural IR.

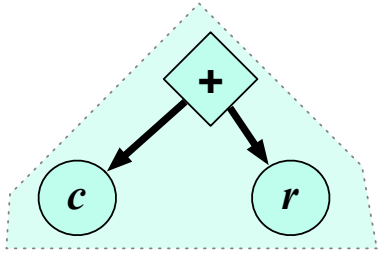
Instruction Selection = Tiling



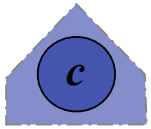

```
add in1, in2 → out
```



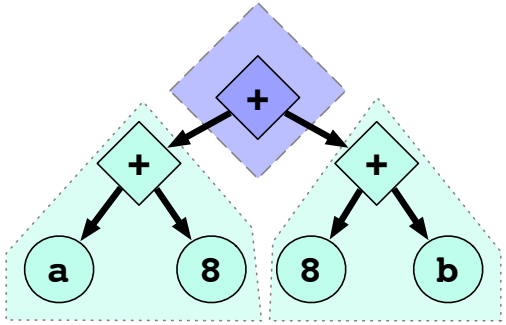

```
add in, reg → out
```




```
add const, reg → out
```




```
move const → out
```

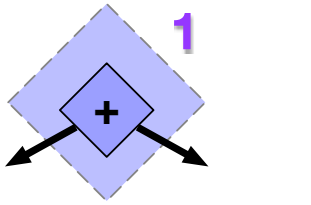



```
add 8, a → t1
add 8, b → t2
add a, b → t3
```

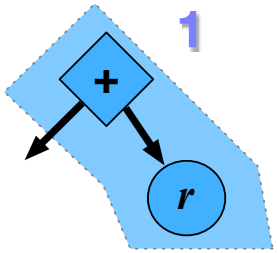
Architecture specific set of *tiles* mapping IR to instructions **+ tiling algorithm =** **instruction selector**

What is the best tiling?

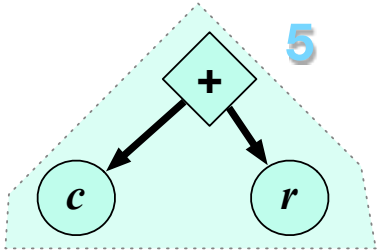
Instruction Selection = Tiling



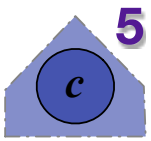

```
add in1, in2 → out
```




```
add in, reg → out
```

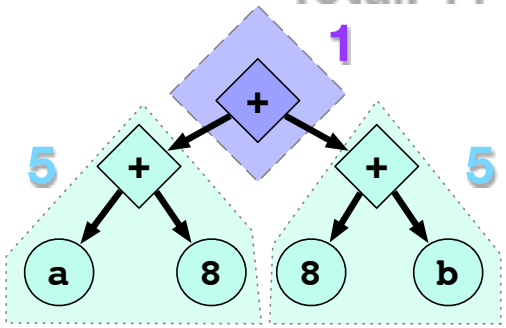



```
add const, reg → out
```




```
move const → out
```

Total: 11




```
add 8, a → t1
add 8, b → t2
add a, b → t3
```

Architecture specific set of *tiles* mapping IR to instructions **+ tiling algorithm =** **instruction selector**

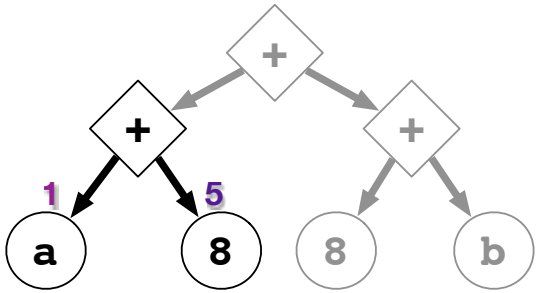
What is the best tiling?

Assign cost to each tile. Minimize cost.

Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

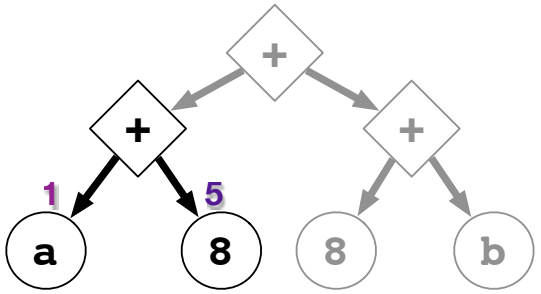
- consider all tiles for the root of the current tree
- sum cost of best subtree tiles and each tile
- choose tile with minimum total cost

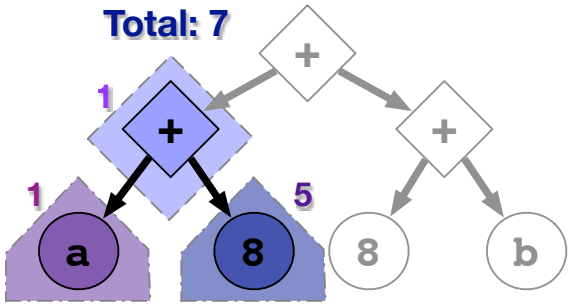


Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

- consider all tiles for the root of the current tree
- sum cost of best subtree tiles and each tile
- choose tile with minimum total cost

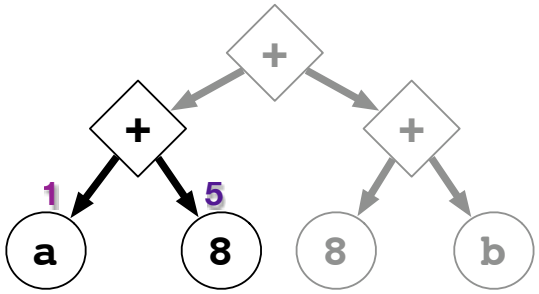


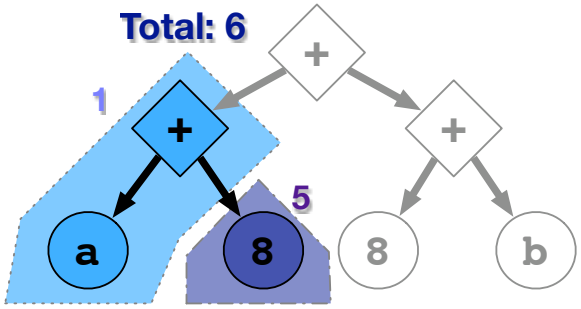


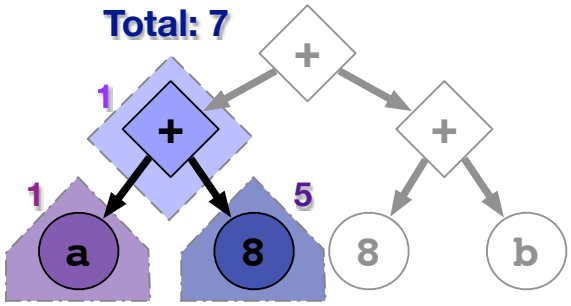
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

- consider all tiles for the root of the current tree
- sum cost of best subtree tiles and each tile
- choose tile with minimum total cost



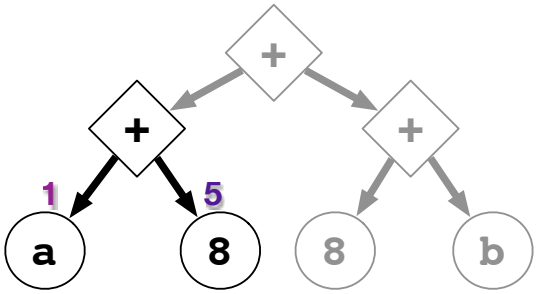


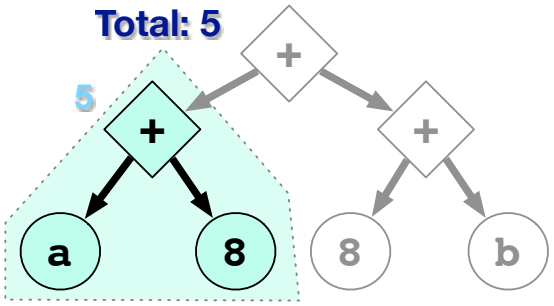


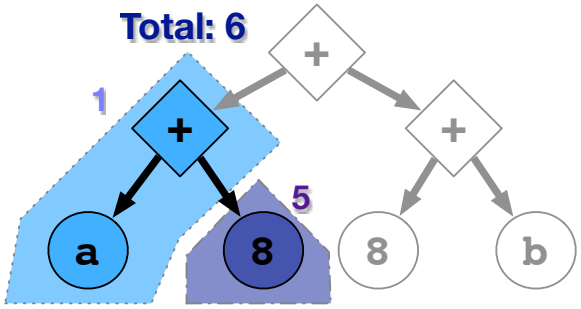
Optimal Tiling on Trees: Bottom Up Dynamic Programming

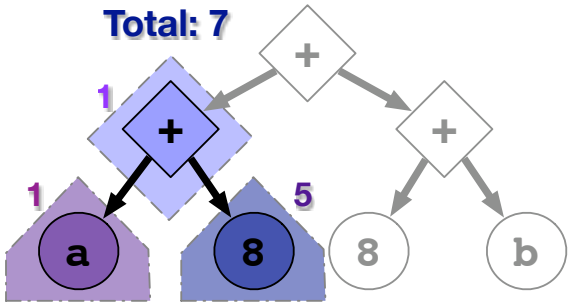
Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

- consider all tiles for the root of the current tree
- sum cost of best subtree tiles and each tile
- choose tile with minimum total cost





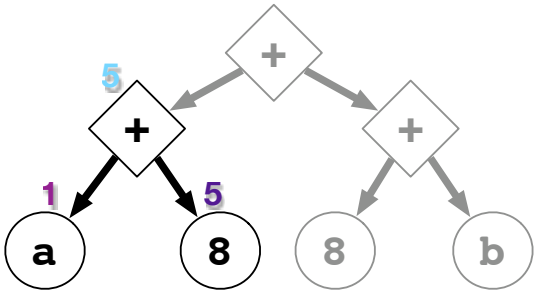


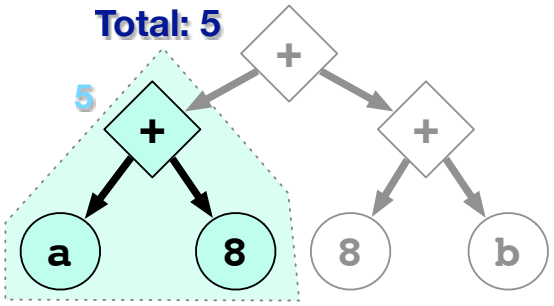


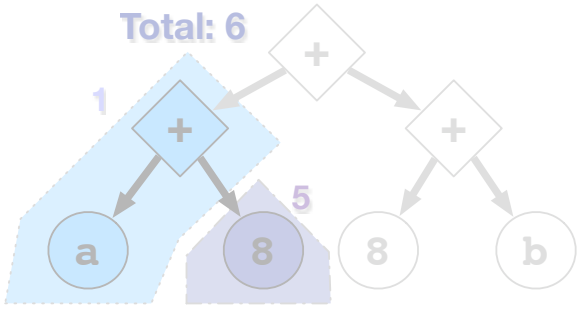
Optimal Tiling on Trees: Bottom Up Dynamic Programming

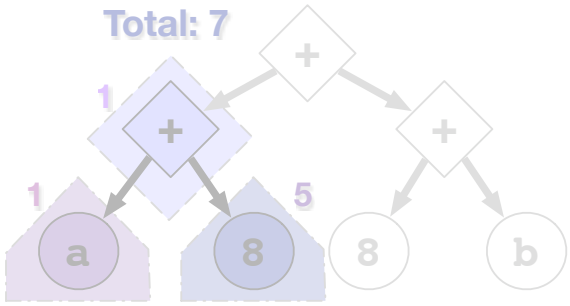
Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

- consider all tiles for the root of the current tree
- sum cost of best subtree tiles and each tile
- choose tile with minimum total cost





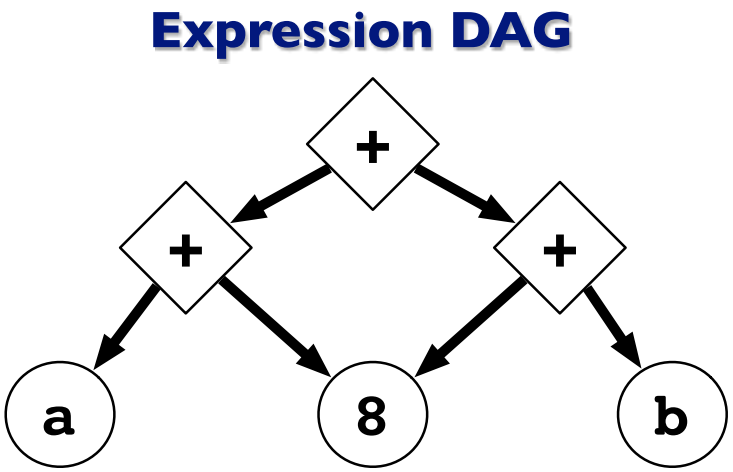




Tiling on Directed Acyclic Graphs

Expression DAGs better representation

– explicitly encode redundant expressions



Tiling on Directed Acyclic Graphs

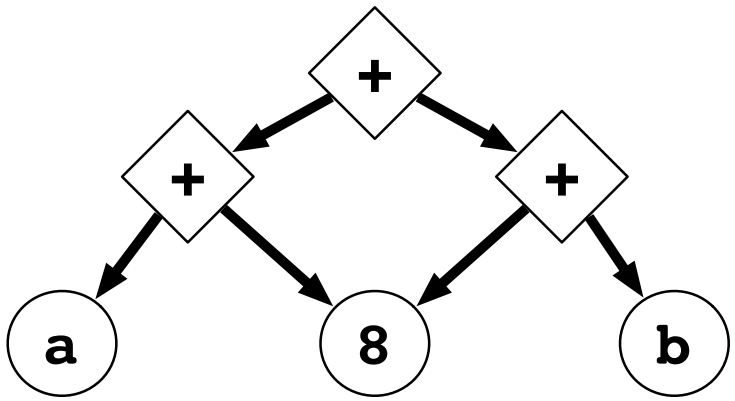
Expression DAGs better representation

– explicitly encode redundant expressions

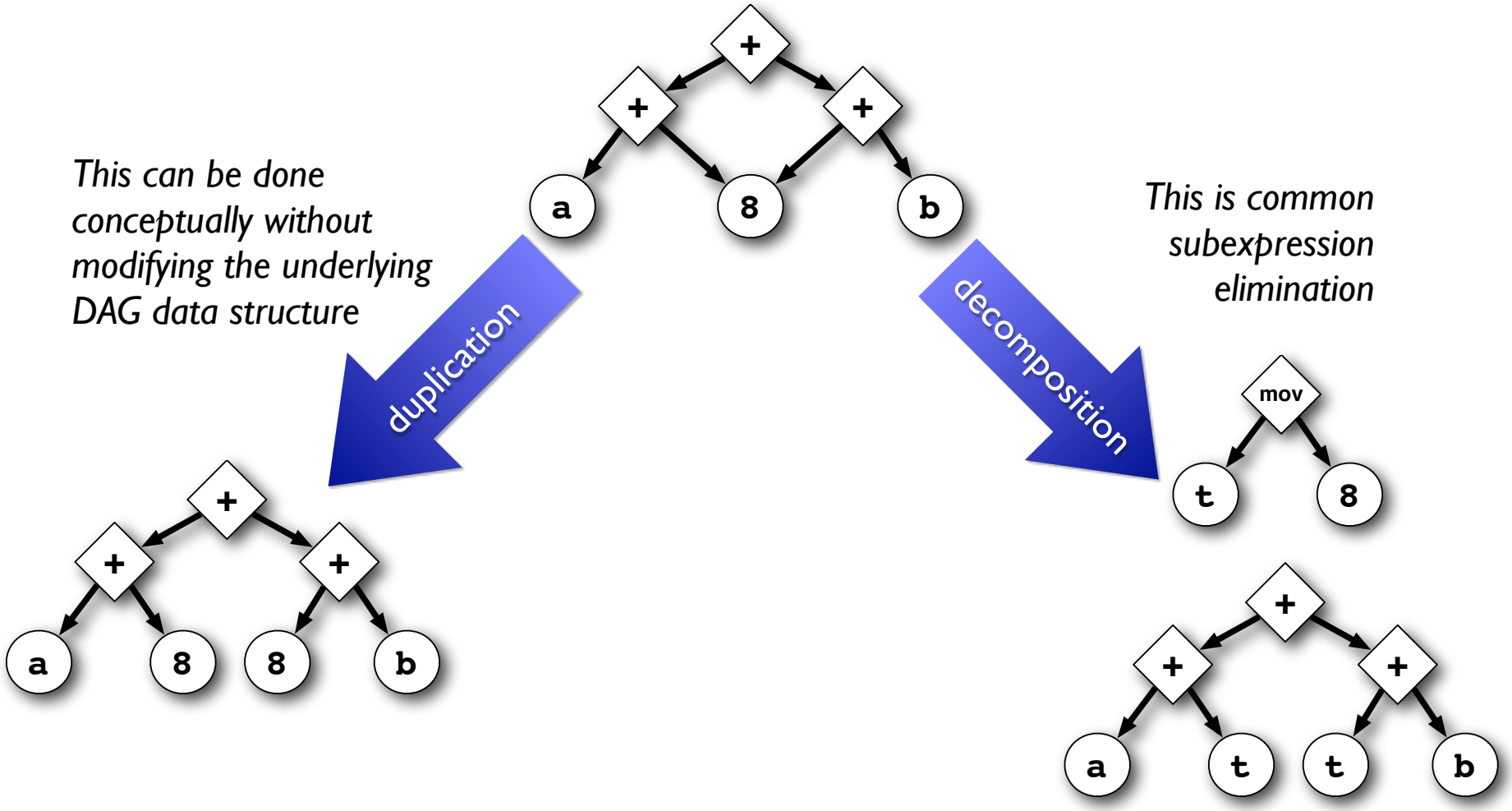
Tiling **NP-complete**

– Heuristic: convert DAG into tree

Expression DAG



Turning a DAG into a Tree



Instruction Selection: State of the Art

Method	DAG Support	Fast	Optimal
Dynamic Programming: Trees	N	Y	Y
Dynamic Programming: DAGS	Y	Y	N
Greedy Matching	Y	Y	N
Peephole Matcher	Y	Y	N
Binate Covering	Y	N	Y

Instruction Selection: State of the Art

Method	DAG Support	Fast	Optimal
Dynamic Programming: Trees	N	Y	Y
Dynamic Programming: DAGS	Y	Y	N
Greedy Matching	Y	Y	N
Peephole Matcher	Y	Y	N
Binate Covering	Y	N	Y
NOLTIS	Y	Y	Nearly

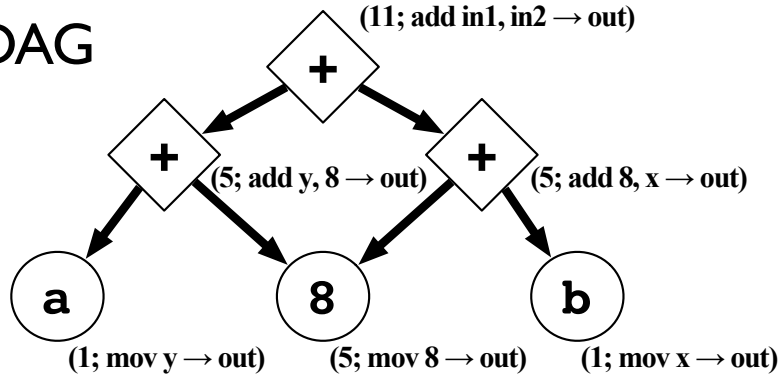
NOLTIS: Near Optimal Linear Time Instruction Selection

1. Run dynamic programming on DAG
   - implicitly duplicate all shared nodes
2. "Fix" shared nodes
   - mark nodes for which decomposition appears more beneficial
3. Rerun dynamic programming
   - "fixed" nodes must be at root of a tile

Dynamic Programming First Pass

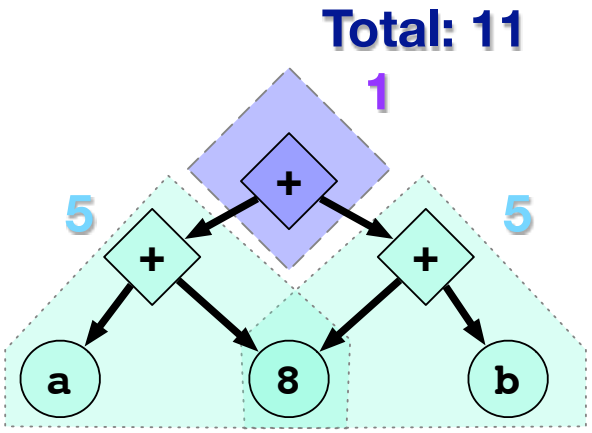
Compute best tiling cost in bottom-up pass

- result is optimal for fully duplicated DAG
- linear time



Obtain tiling in top-down pass

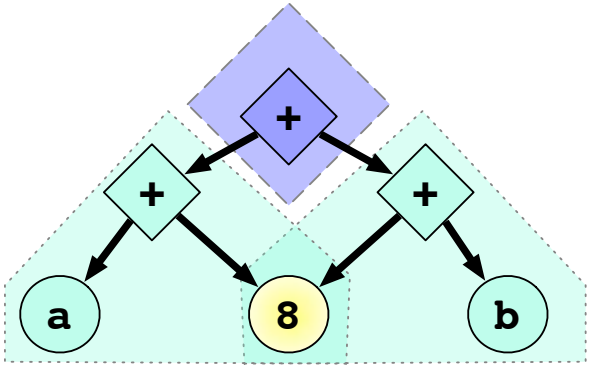
- avoid redundant overlap
- linear time



Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

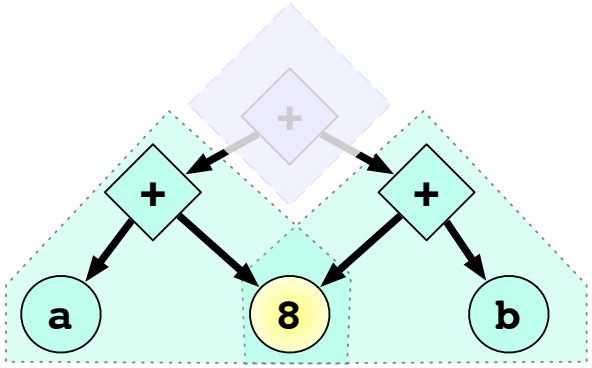
- assuming rest of tiling remains the same, what happens to the cost if we "cut" the tiles overlapping this shared node?
- if cost improves, "fix" the node



Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

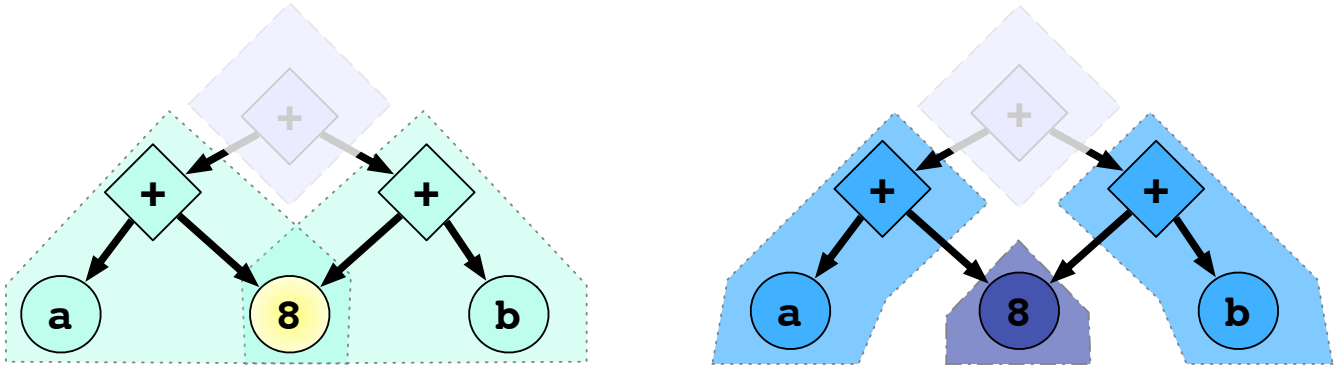
- assuming rest of tiling remains the same, what happens to the cost if we "cut" the tiles overlapping this shared node?
- if cost improves, "fix" the node



Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

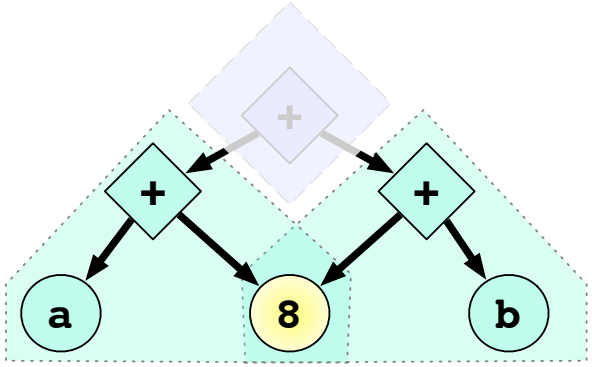
- assuming rest of tiling remains the same, what happens to the cost if we "cut" the tiles overlapping this shared node?
- if cost improves, "fix" the node



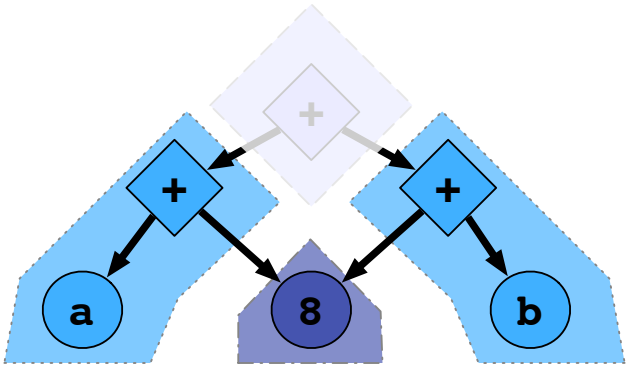
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

- assuming rest of tiling remains the same, what happens to the cost if we "cut" the tiles overlapping this shared node?
- if cost improves, "fix" the node



cost = 5 + 5 = 10

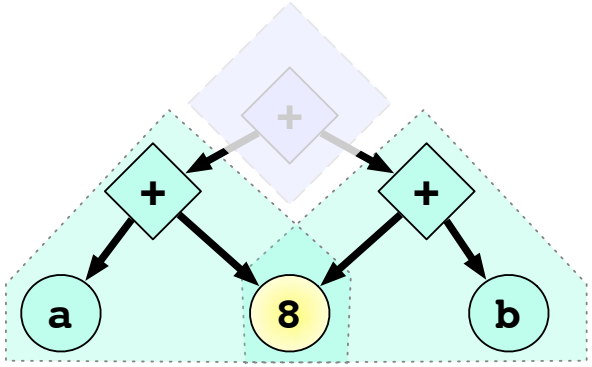


cost = 5 + 1 + 1 = 7

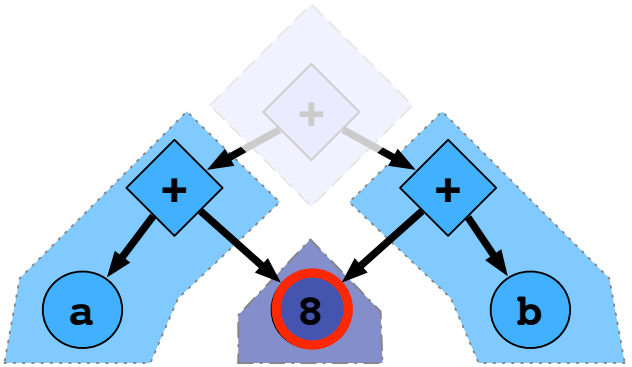
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

- assuming rest of tiling remains the same, what happens to the cost if we "cut" the tiles overlapping this shared node?
- if cost improves, "fix" the node



cost = 5 + 5 = 10

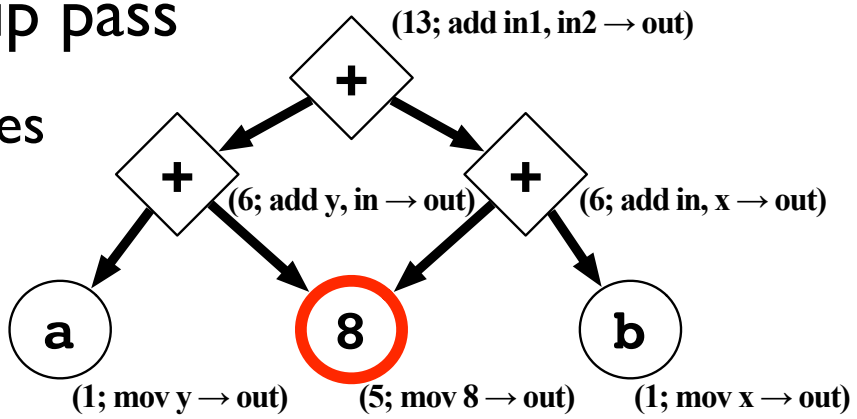


cost = 5 + 1 + 1 = 7

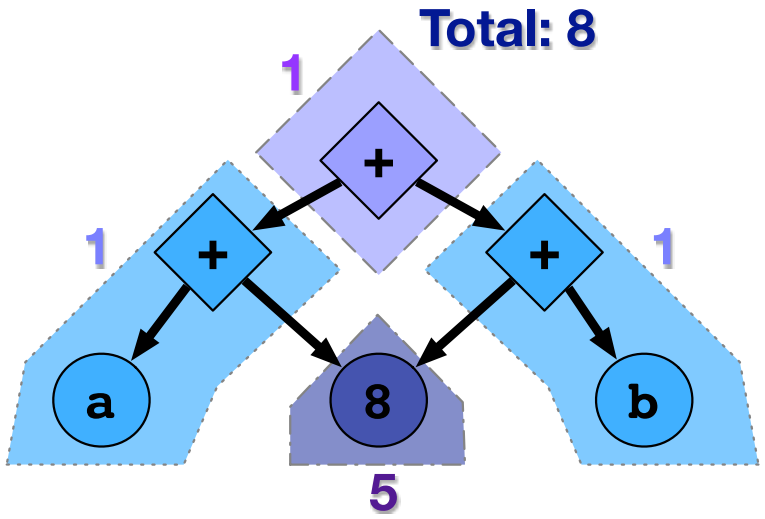
Dynamic Programming Second Pass

Compute best tiling in bottom-up pass

– tiles not allowed to span fixed nodes



Obtain tiling in top-down pass



NOLTIS Implementation

LLVM 2.1 compiler infrastructure targeting x86

Algorithms implemented:

default	greedily select largest tile in top-down topological traversal of DAG
cse-all	decompose entire DAG into trees then perform dynamic programming
cse-leaves	decompose non-leaf expressions into trees, duplicate leaf expressions and perform dynamic programming
cse-none	perform dynamic programming on DAG treating shared nodes as duplicated
NOLTIS	near optimal linear-time instruction selection

Evaluating NOLTIS: Optimality

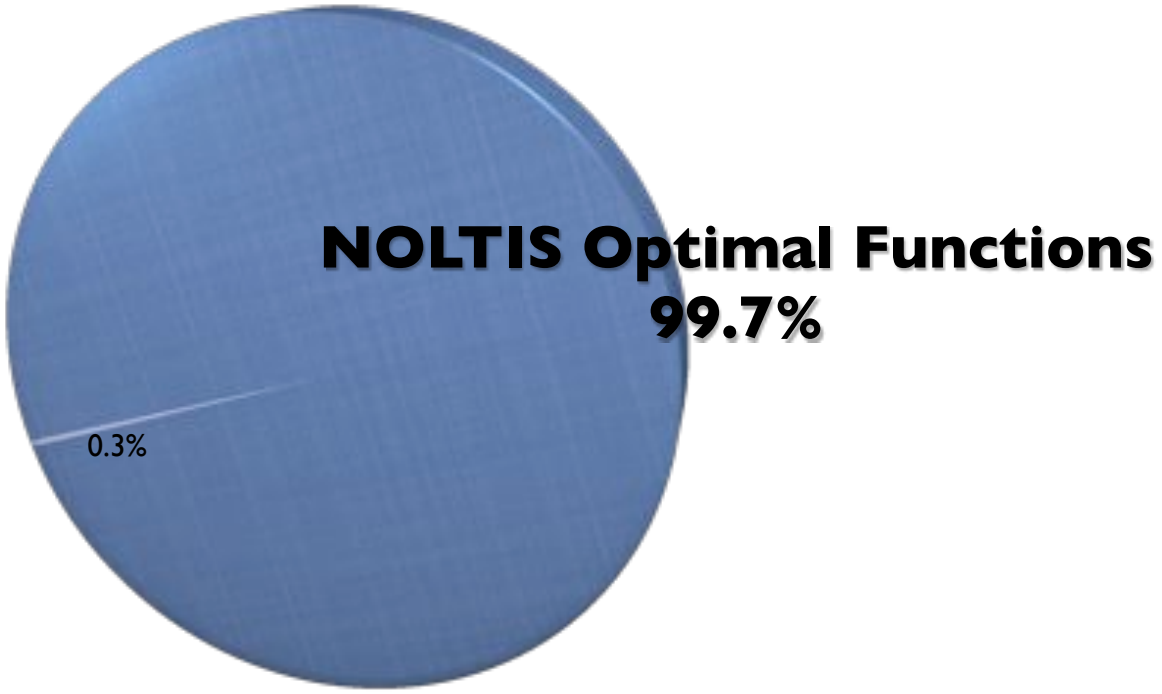
Compute optimal instruction tiling using integer linear programming and ILOG CPLEX 10.0

Evaluated nearly half a million functions

Evaluating NOLTIS: Optimality

Compute optimal instruction tiling using integer linear programming and ILOG CPLEX 10.0

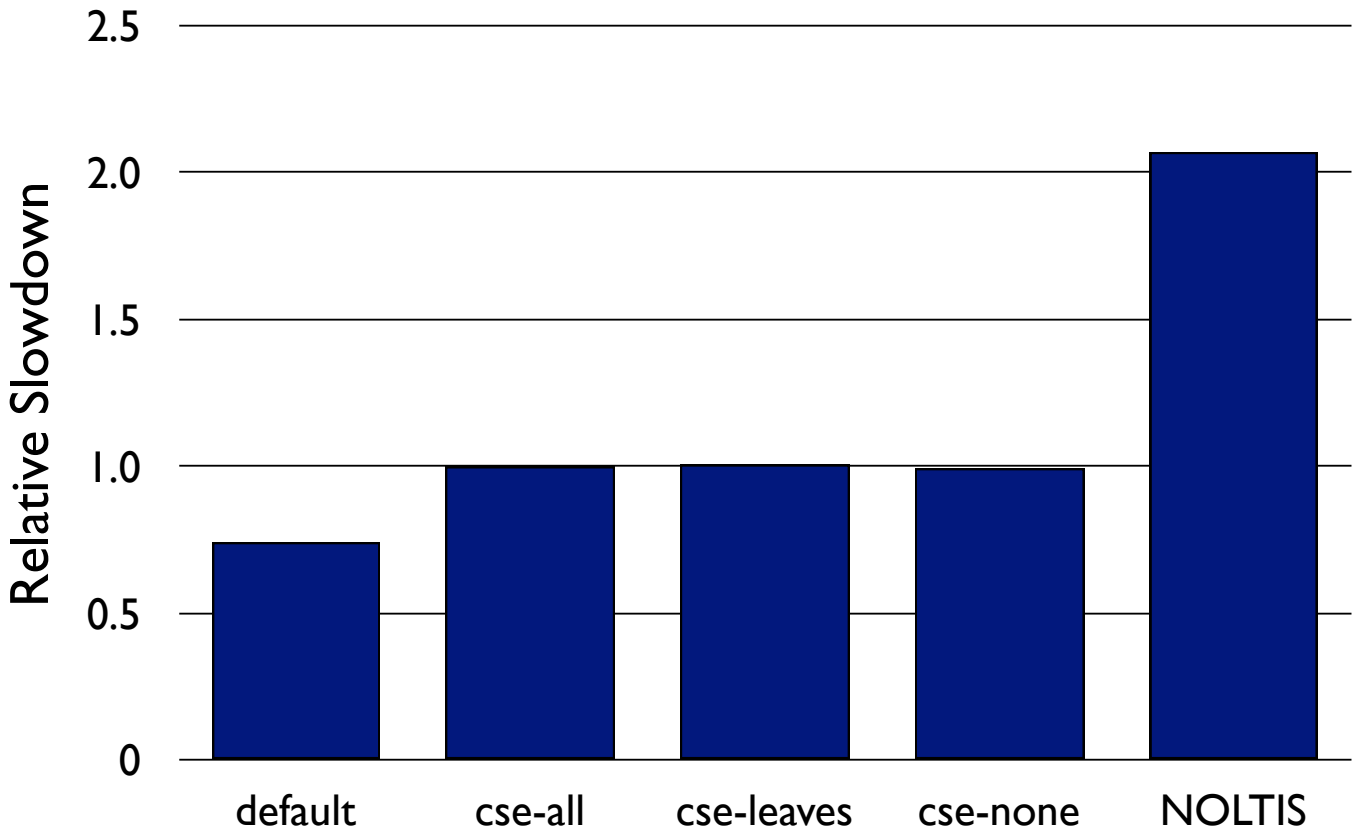
Evaluated nearly half a million functions



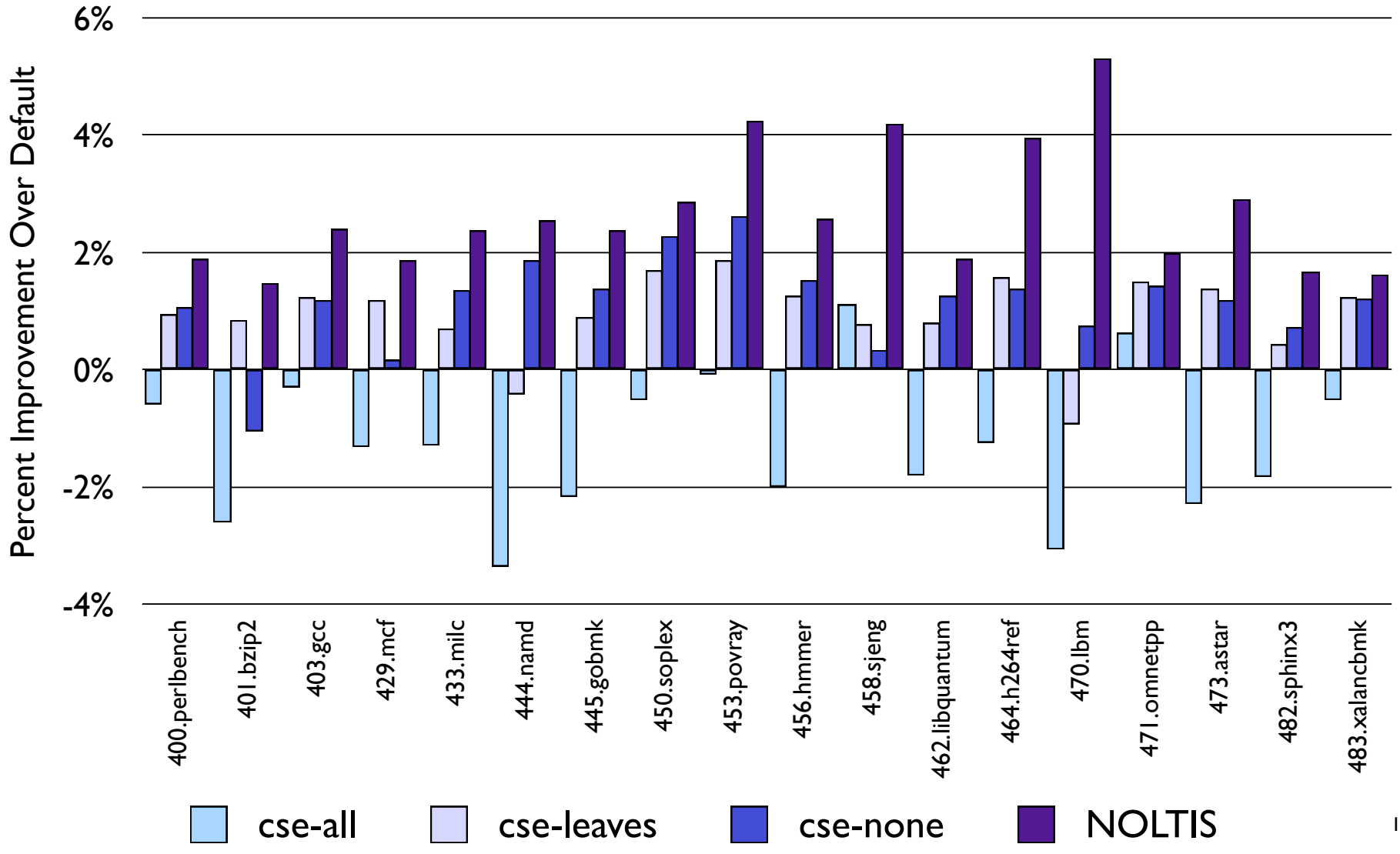
Evaluating NOLTIS: Compile Time

Two pass algorithm results in 2X slowdown

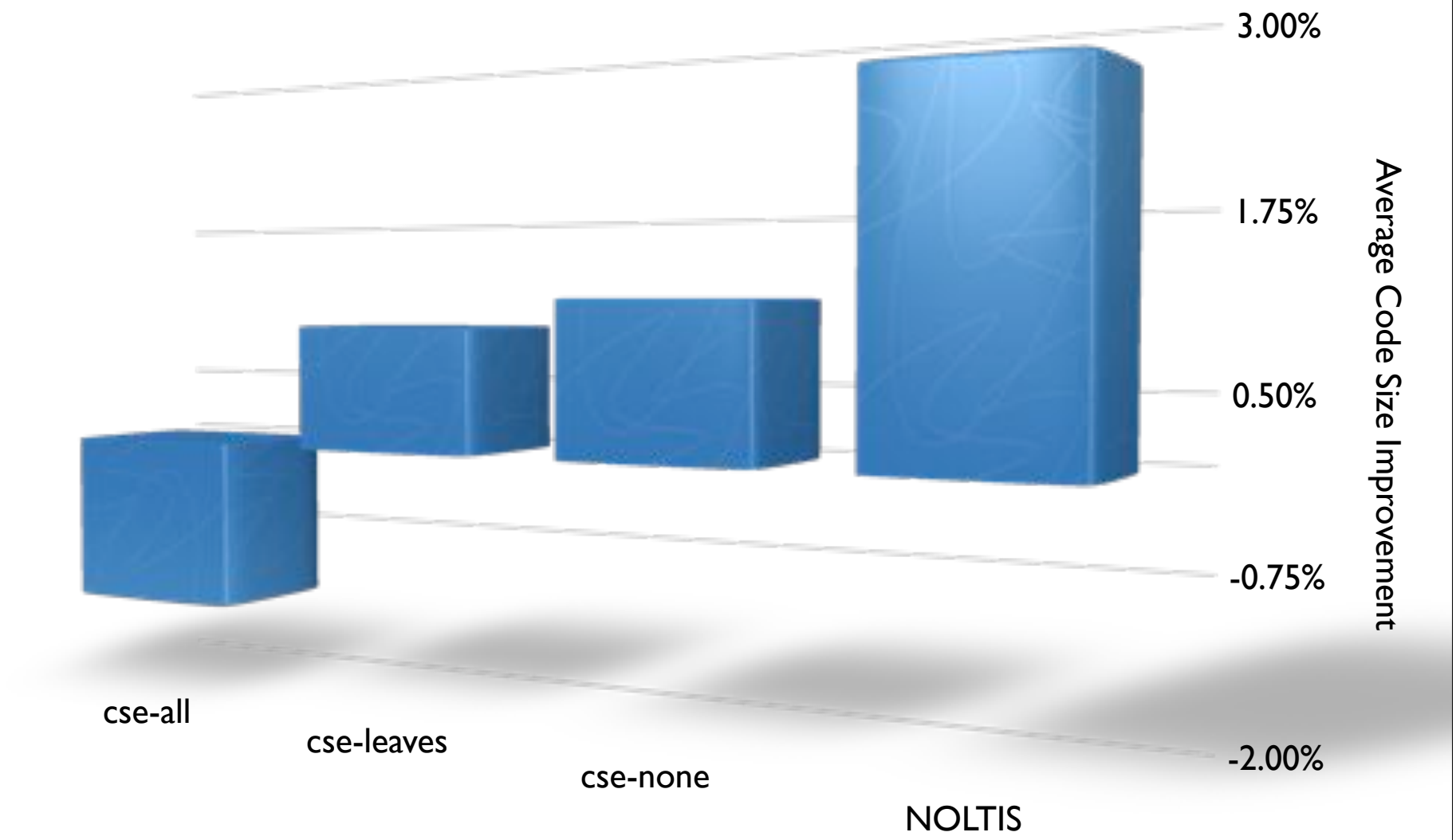
– each linear time pass is ideally a small part of total compile-time



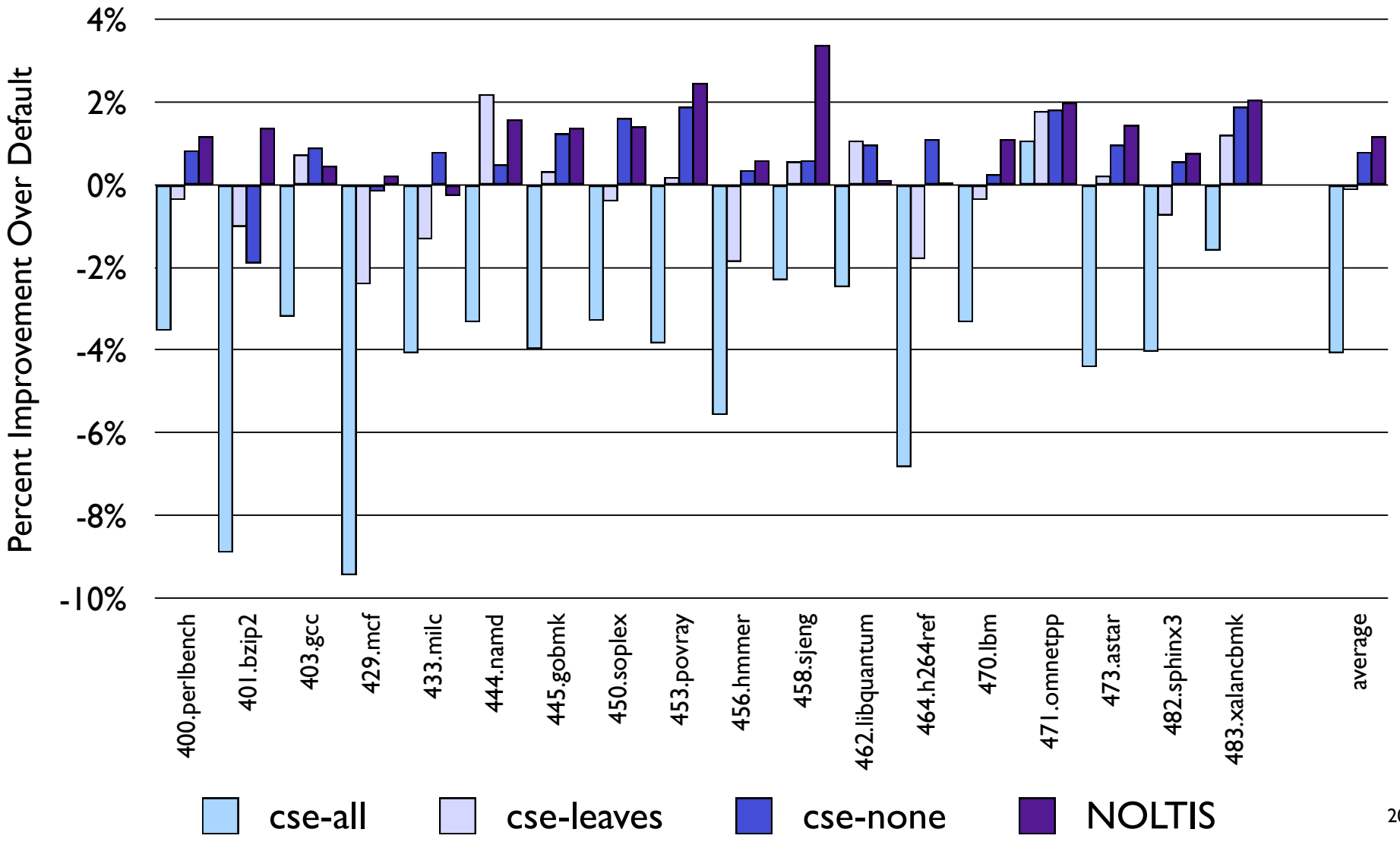
Evaluating NOLTIS: Code Size After Instruction Selection



Evaluating NOLTIS: Code Size After Instruction Selection



Evaluating NOLTIS: Final Code Size



Conclusions

NOLTIS is fast, effective, and easy to implement

Expression DAGs are better than trees

But, need to further investigate interaction between instruction selection and register allocation

Conclusions

NOLTIS is fast, effective, and easy to implement

Expression DAGs are better than trees

But, need to further investigate interaction between instruction selection and register allocation

My thesis topic!

Conclusions

NOLTIS is fast, effective, and easy to implement

Expression DAGs are better than trees

But, need to further investigate interaction between instruction selection and register allocation

My thesis topic!

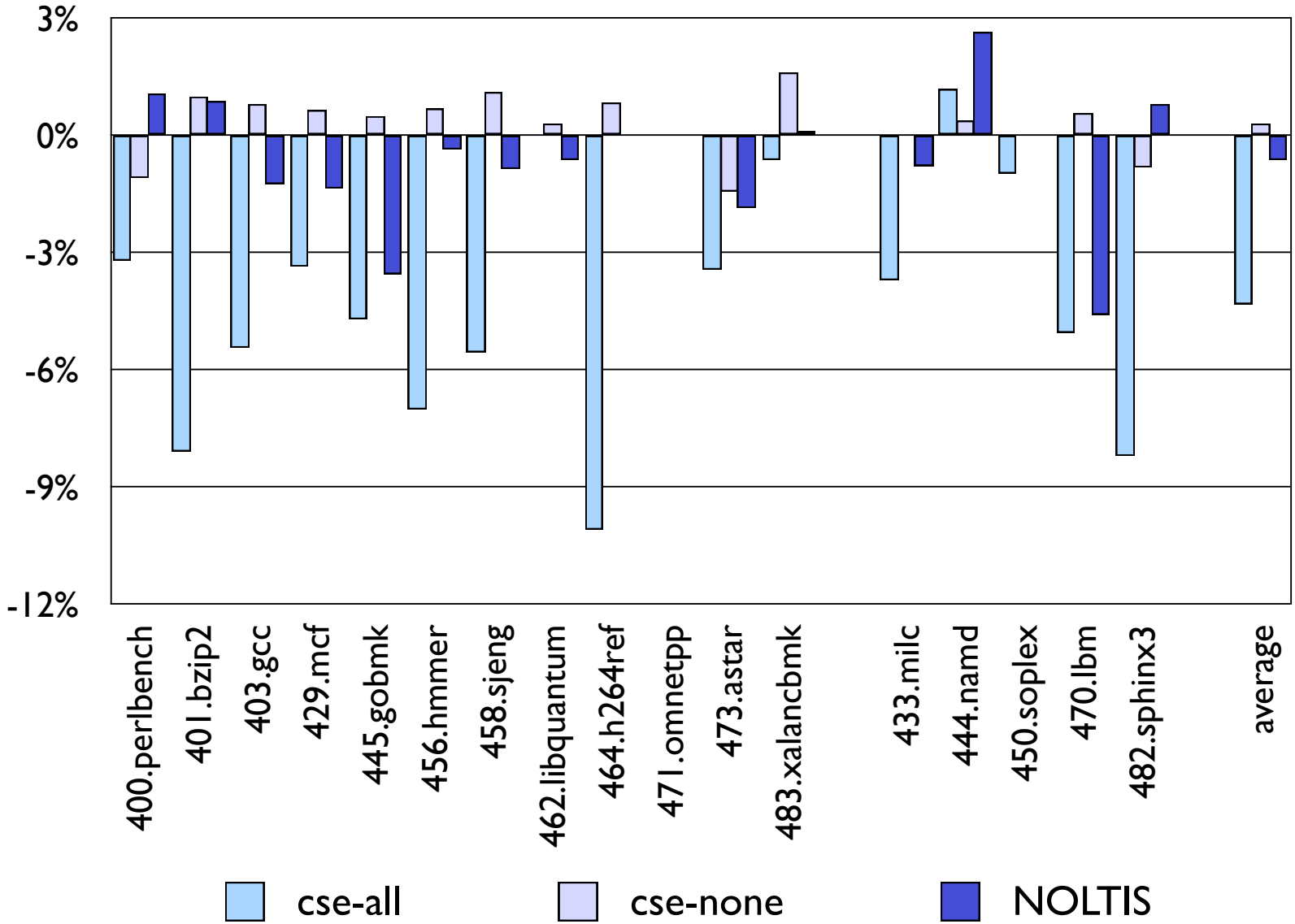
Questions?

http://www.cs.cmu.edu/~dkoes

Questions?

http://www.cs.cmu.edu/~dkoes

Performance Improvement

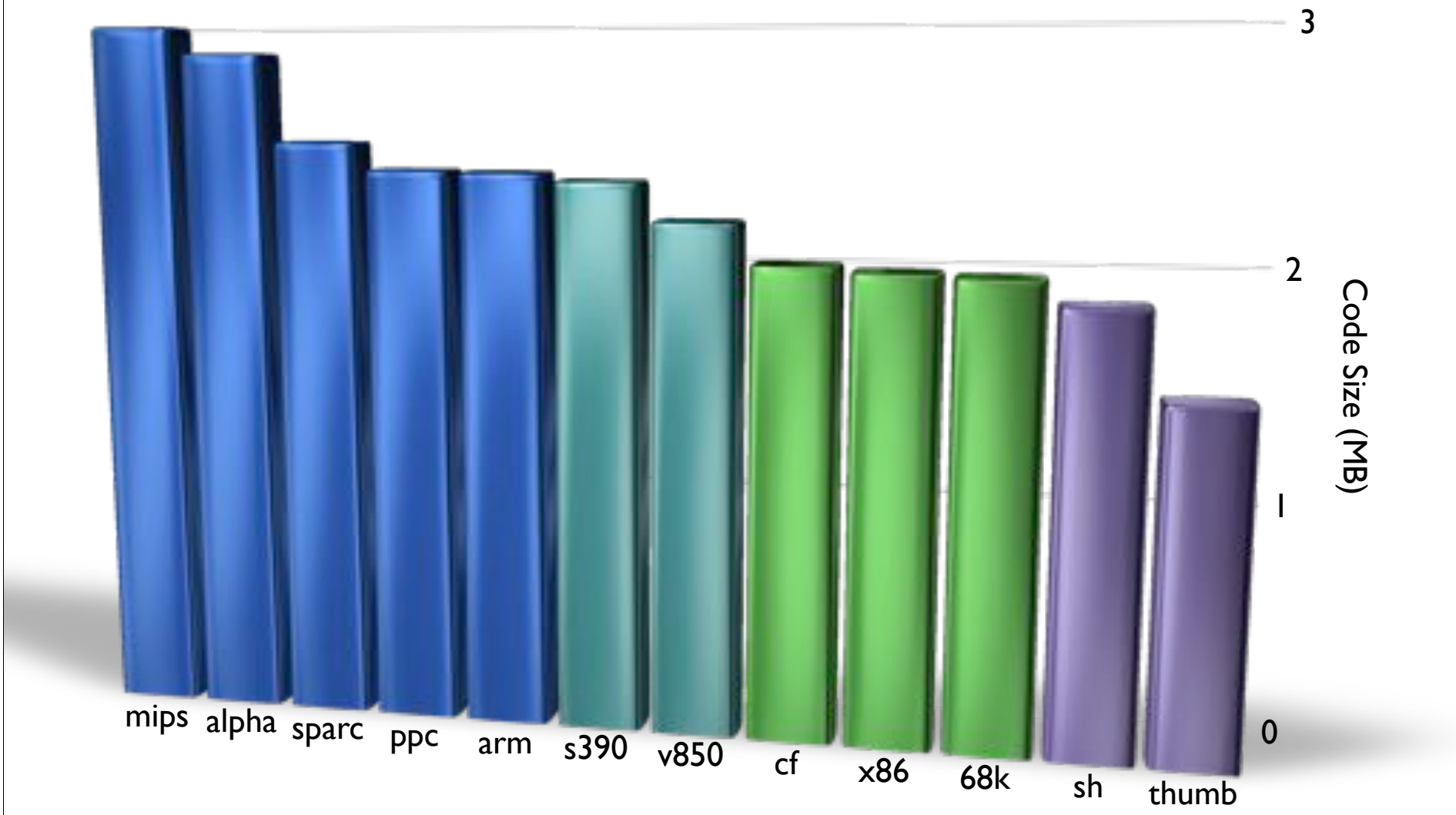


Impact of ISA on Code Size

Architecture	Instruction Size	Integer Registers	FP Registers
68k (68040)	2-14	16 (8/8)	8
Alpha	4	32	32
Arm	4	16	8
Arm Thumb	2	8*	
Coldfire (V4e)	2, 4, 6	16 (8/8)	8
MIPS32	4	32	32
NEC v850	2, 4	32	0
PowerPC (750)	4	32	32
s390	2, 4	16	16
Sparc	4	32	32
SuperH (SH4)	2	16	16+16
x86	1-15	8	8

*Additional registers can be accessed inefficiently

Impact of ISA on Code Size



Results obtained using gcc 4.2.1 compiling the 403.gcc benchmark of SPEC2006 using the -Os option