Compiling for the Multiscalar Architecture

T.N. Vijaykumar

PhD Defense

Computer Sciences Department

University of Wisconsin-Madison

Background

Microprocessors are compute engines for a range of computers

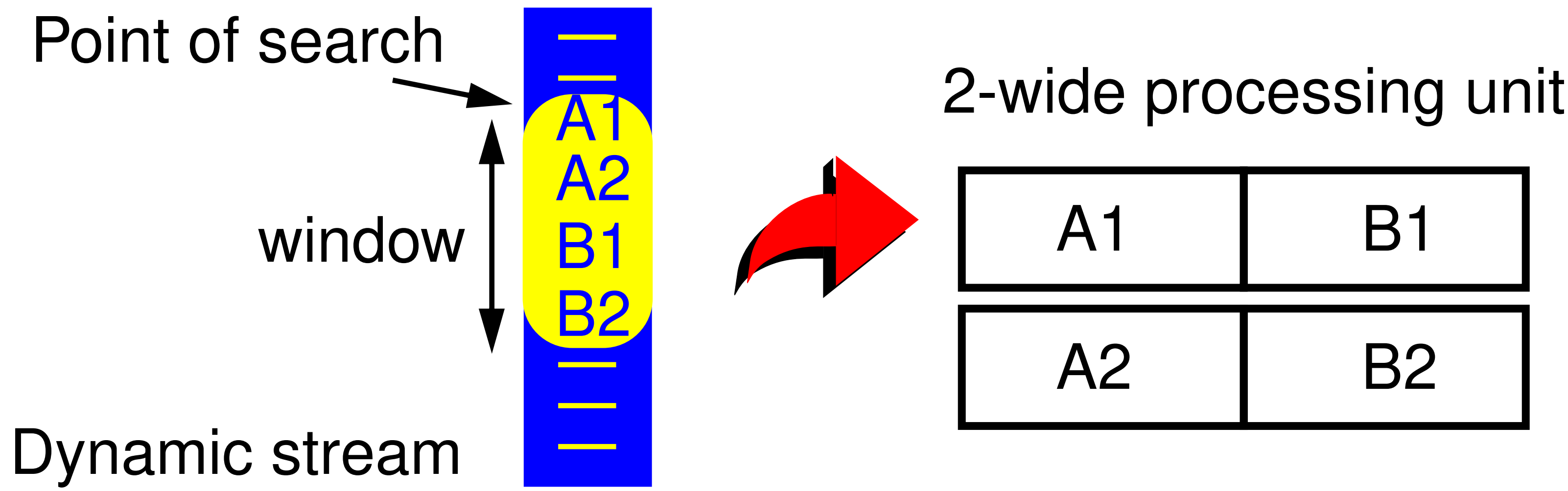
Applications: sequential programs

High performance is important

Exploit Instruction Level Parallelism to achieve performance

Exploiting ILP: State of the art

Search for independent instructions in a window

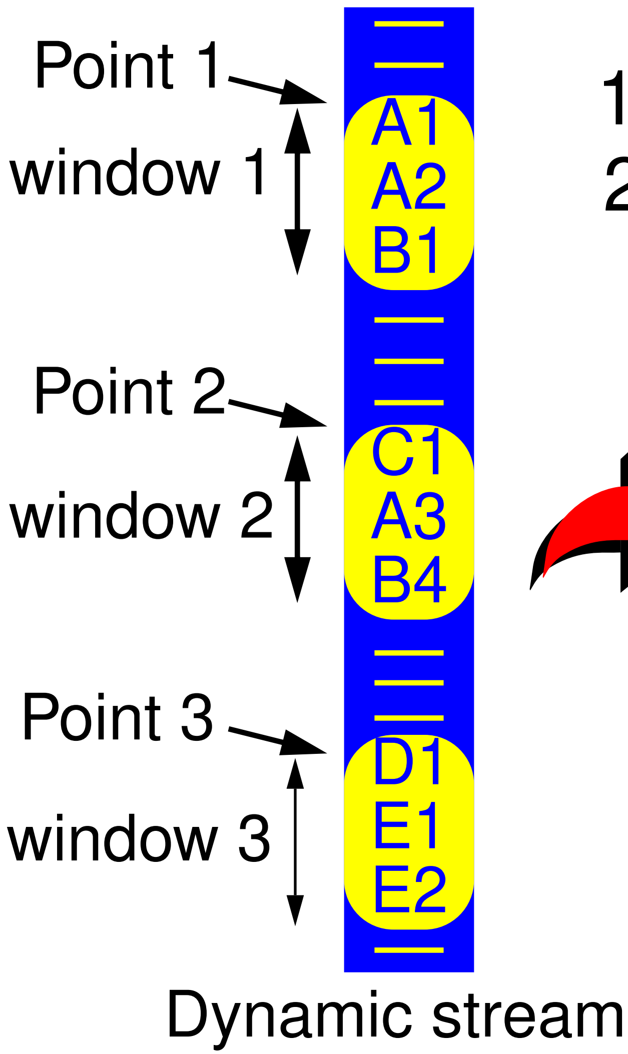


+ Larger window, wider machine - more parallelism

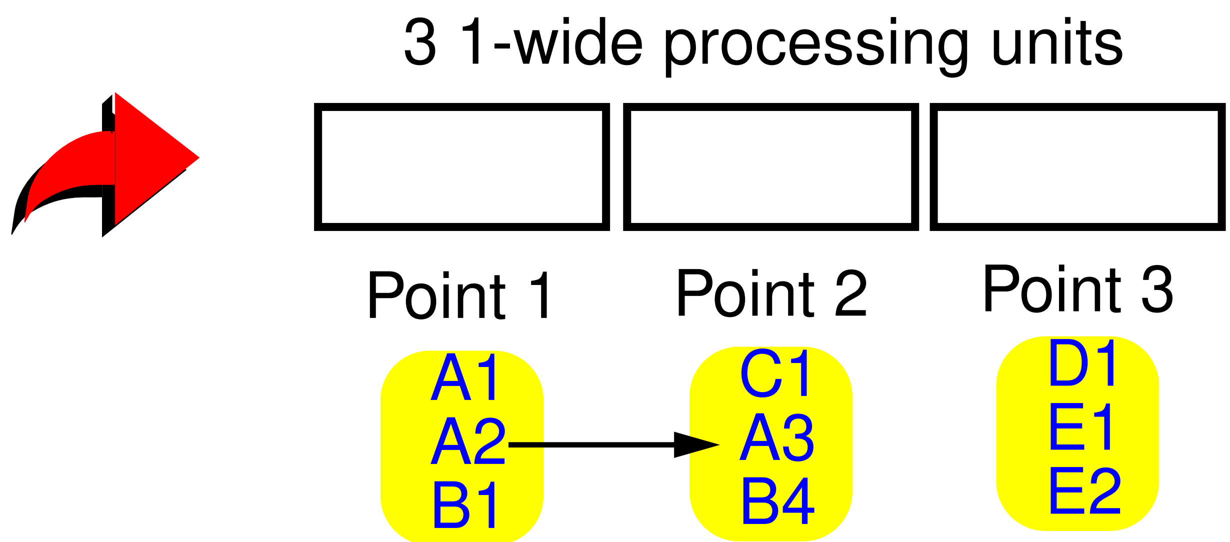
– Larger window, wider machine - hard to clock fast (21264)

Typical window size: 60-100, machine width: 4-8

More Parallelism and Fast Clock?



1 Search at multiple points
2 Split big window and wide machine



Multiscalar Architecture: Who does what?

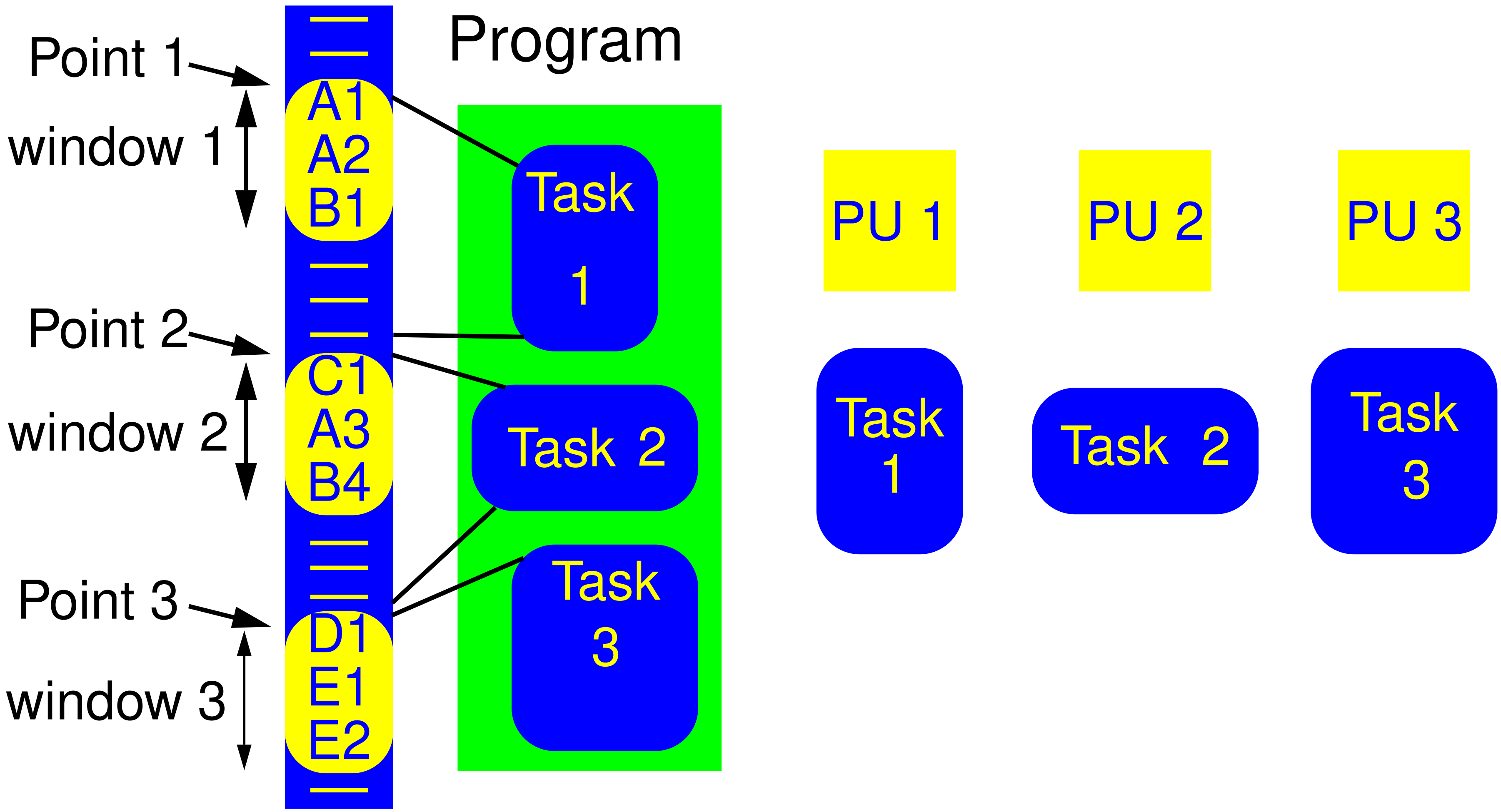
In the implementation that I considered

Deciding the points of search: compiler

Maintaining original program dependencies across little windows

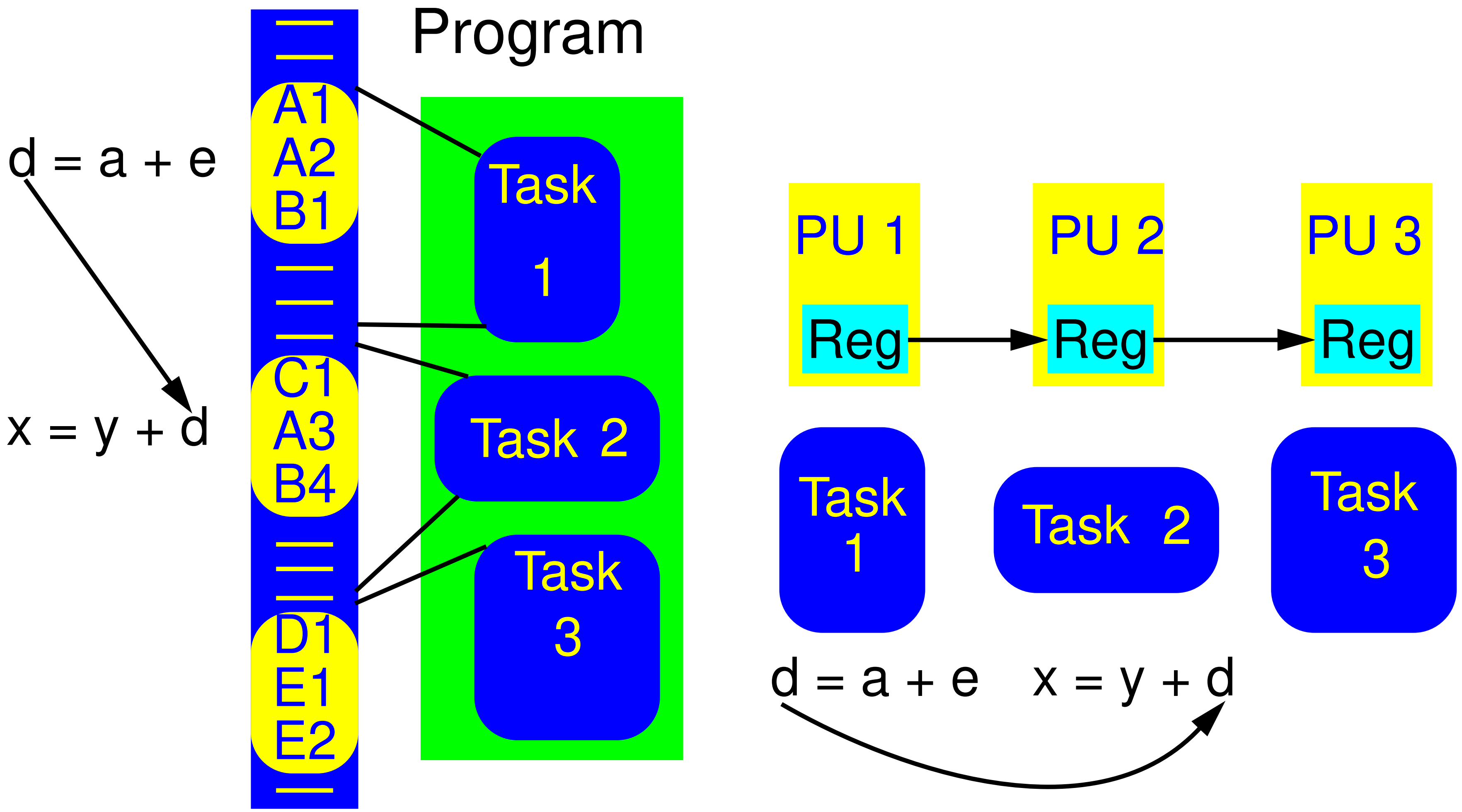
- Register dependencies: compiler
- Control flow: hardware
- Memory dependencies: hardware

Role of the Compiler: Task Selection



Compiler decides the points of search by partitioning programs

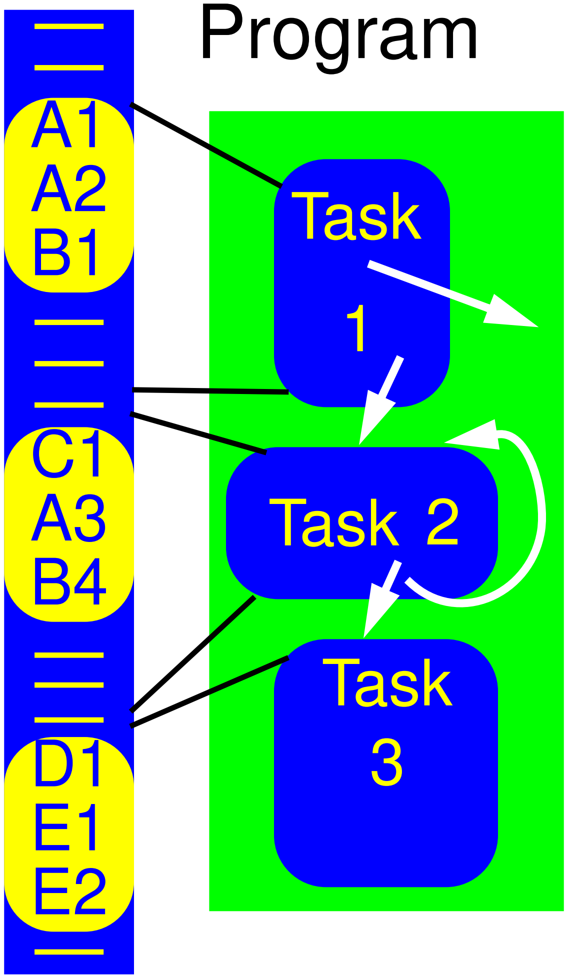
Role of the Compiler: Register Values



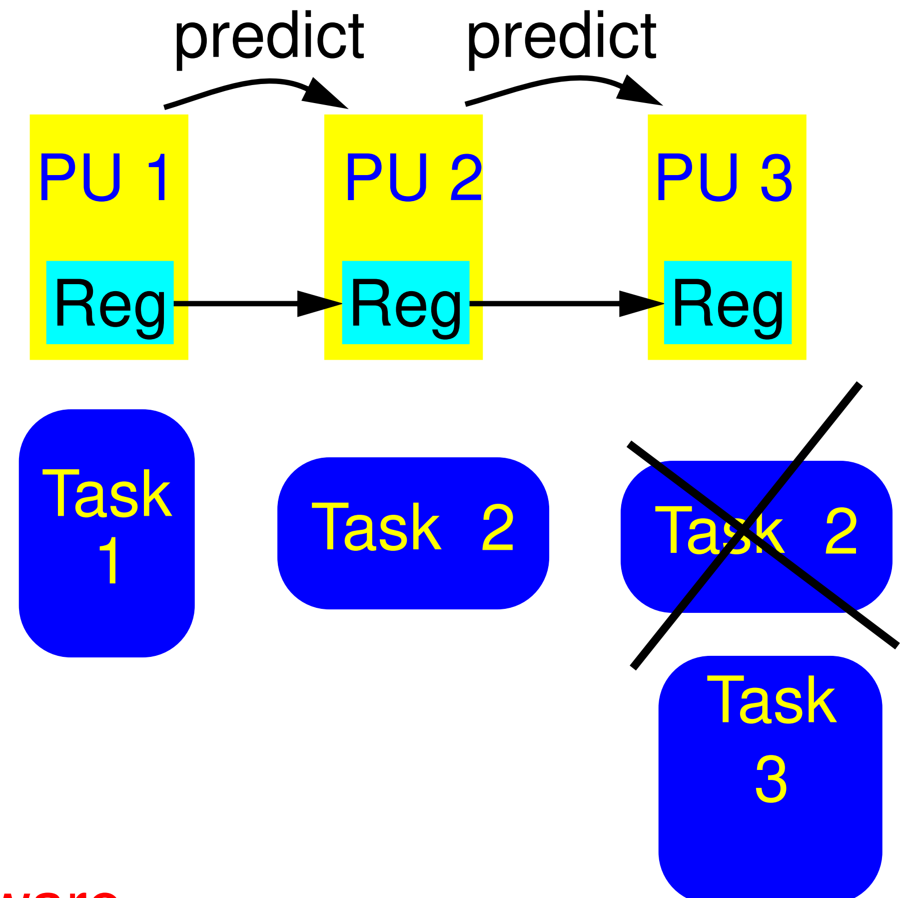
Compiler orchestrates register value communication

Role of the Hardware: Control Flow

Program

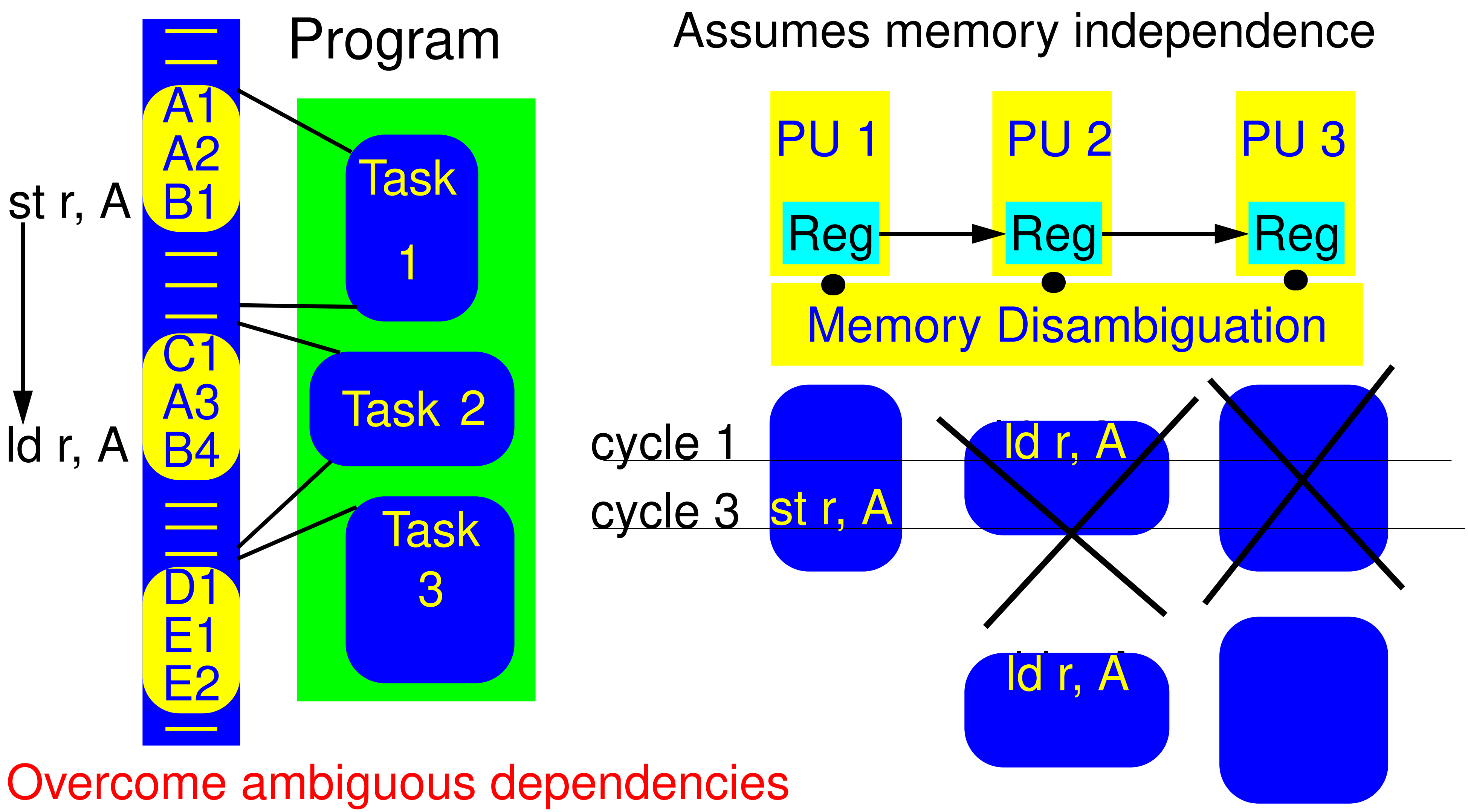


Hardware predicts one of the exits



Misspeculation rollback in hardware

Role of the Hardware: Memory Values



Overcome ambiguous dependencies

Contributions

Studied interaction between programs and architecture

Identified fundamental performance issues

Constructed a compiler for the Multiscalar architecture

- Partition sequential programs into tasks
- Generate inter-task register communication
- Schedule register communication
- Loop restructuring, dead register optimizations
- Specify program information (register and control)

Evaluated compiler techniques using a simulator for Multiscalar

Overview of the Compiler

Traditional (Gcc)

- Parsing
- Jump Optimization
- Common Sub-Expression
- Loop Optimization
- Register Allocation
- Code Generation

Multiscalar

- Loop Restructuring
- Task Selection
- Register Communication Scheduling
- Task Annotation

Roadmap

~~Introduction~~

Task Selection

Inter-task register communication scheduling

Summary

Multiscalar Tasks

Connected, single entry subgraph of the CFG

Corresponds to a contiguous fragment of the dynamic stream

Basic block, multiple basic blocks, loops, function invocations

Arbitrary control and data dependences

Wide spectrum of choices

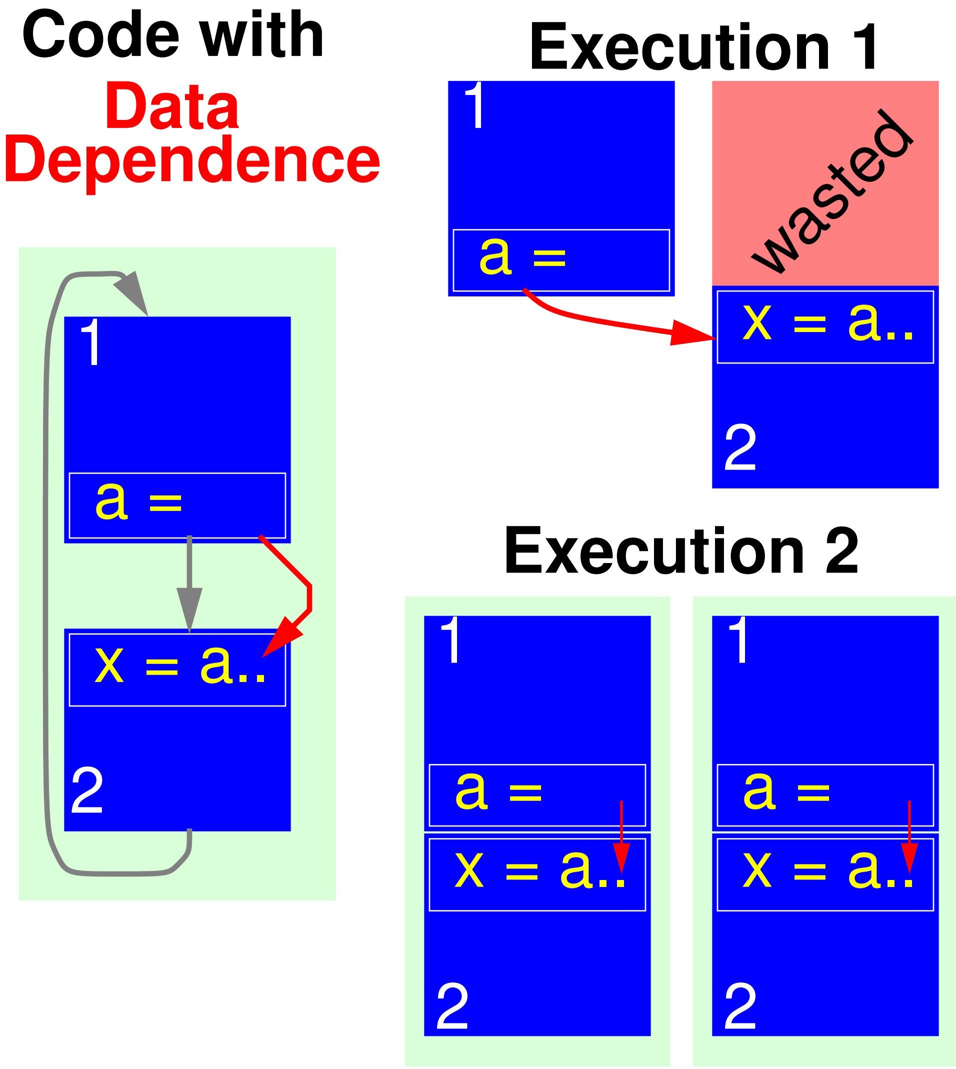
Task Selection: Factors

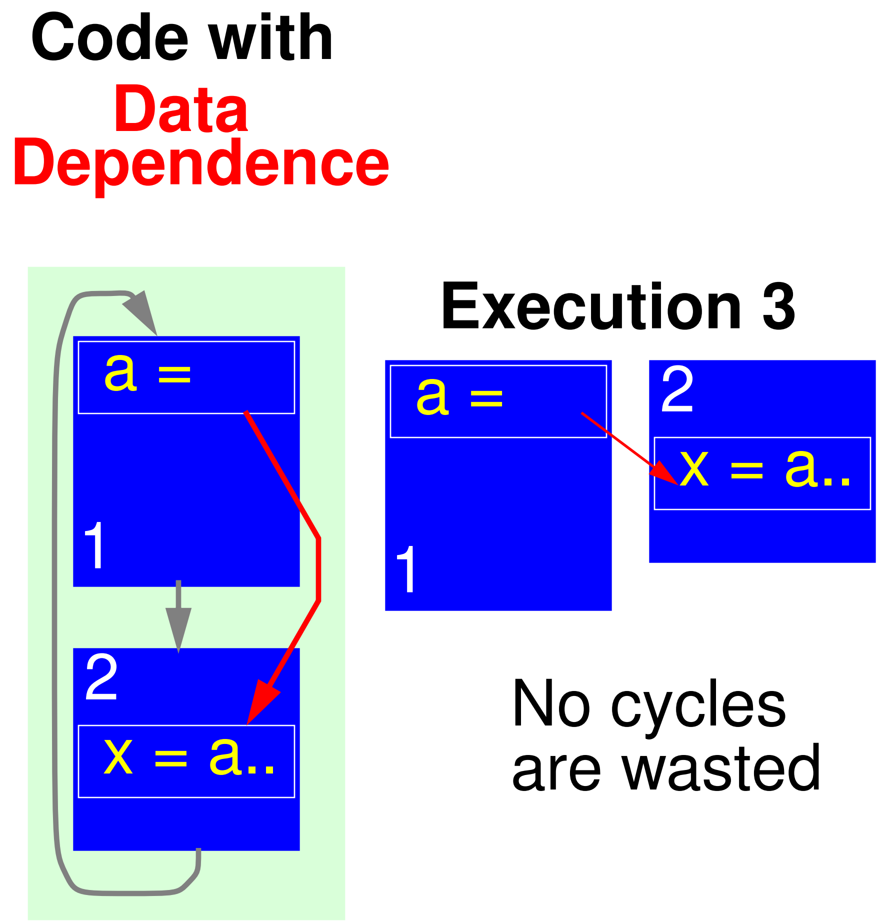
How do tasks impact performance?

Fundamental factors interact with task selection

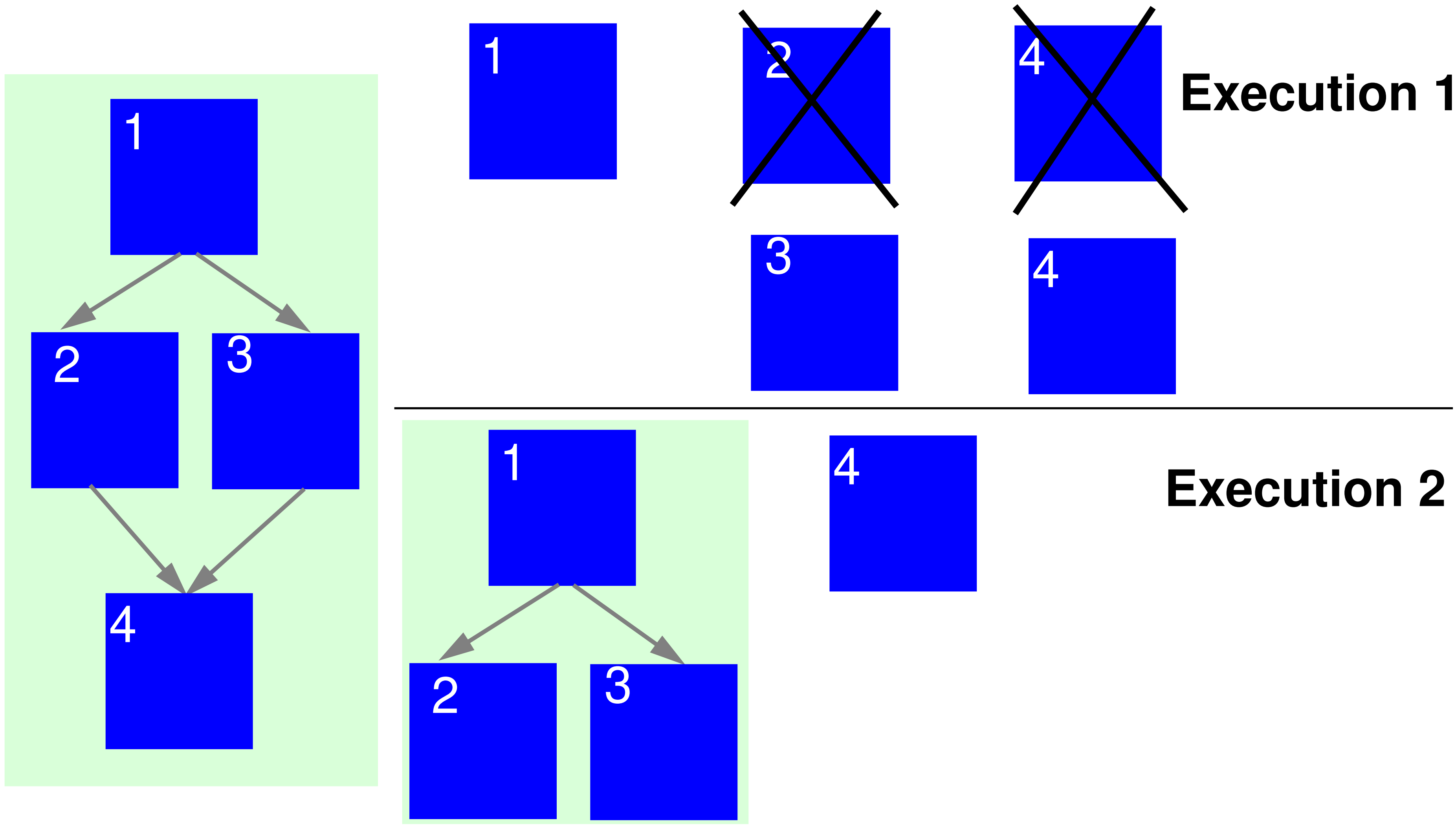
- Inter-task data dependences
- Inter-task control flow
- Task overheads
- Load imbalance

Inter-task Data Dependence

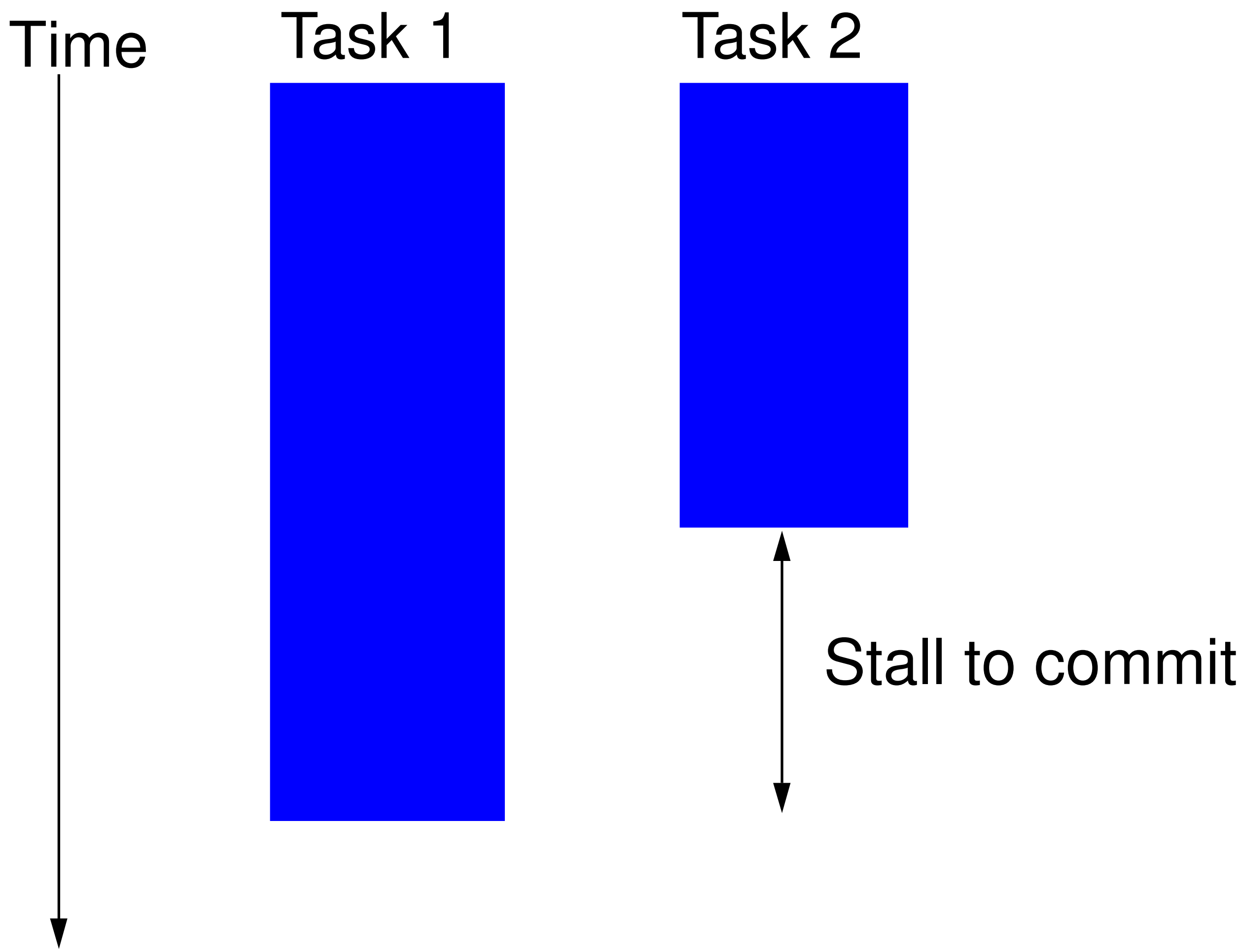




Inter-task Control Flow



Load Imbalance



Task Size

Small

- Task start/end overheads
- Low degree of parallelism

Large

- Lost opportunity to exploit the parallelism within
- More likely to cause memory misspeculations
- Speculative buffers may overflow

Variation in size

- Load imbalance

Task Selection

Involves many complex factors

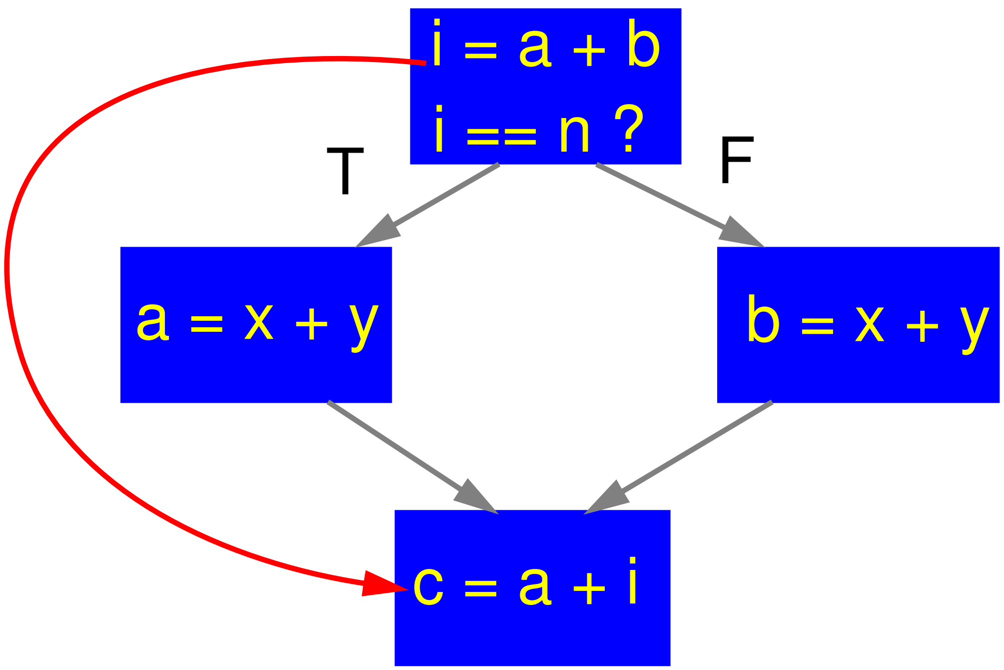
NP-Complete [Sarkar for functional programs on Multiprocessors]

Viewed as partitioning the Control Flow Graph (CFG) of program

```
i = a + b

if (i == n)
   a = x + y
else
   b = x + y

c = a + i
```



Heuristics: Control Flow

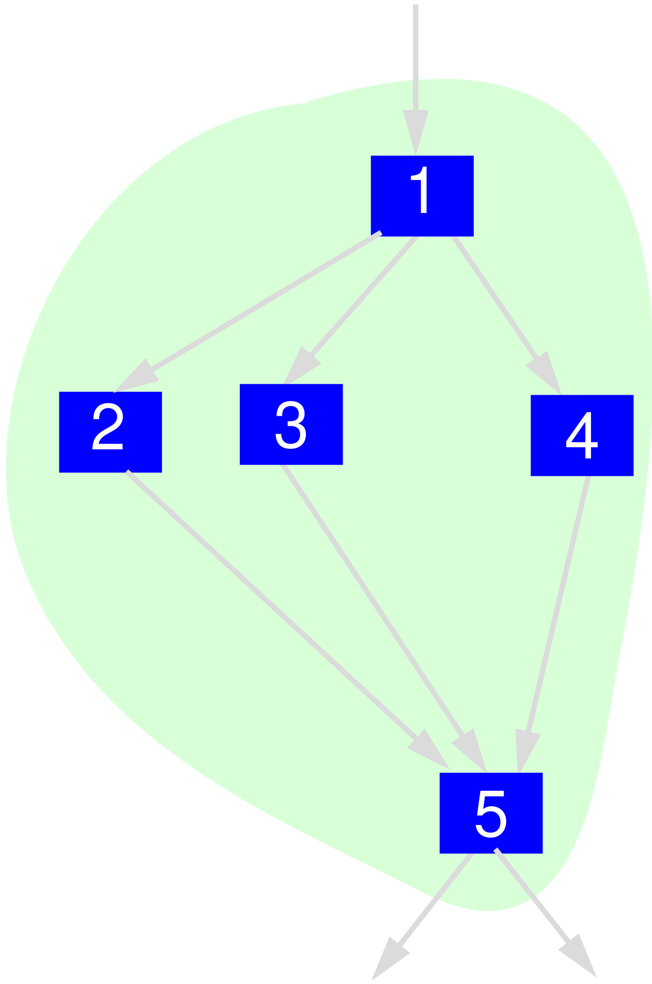
Include multiple basic blocks

Take advantage of reconvergent control flow paths

Control number of successors

Graph traversal of the CFG with greedy selection

Heuristics: Control Flow



Heuristics: Task Size

Terminate at loop back edges, function invocations

Suppress short function invocations

Unroll short loops

Heuristics: Data Dependence

Try to “include” data dependencies within tasks

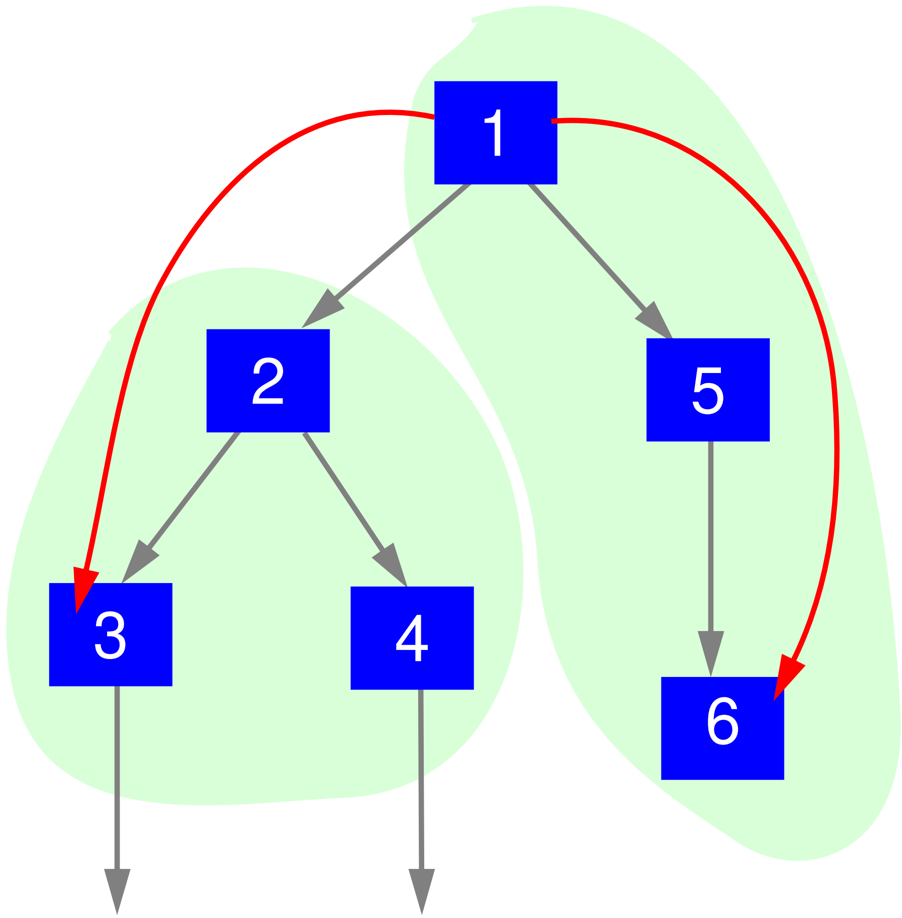
But including one may exclude another

Prioritize with frequency

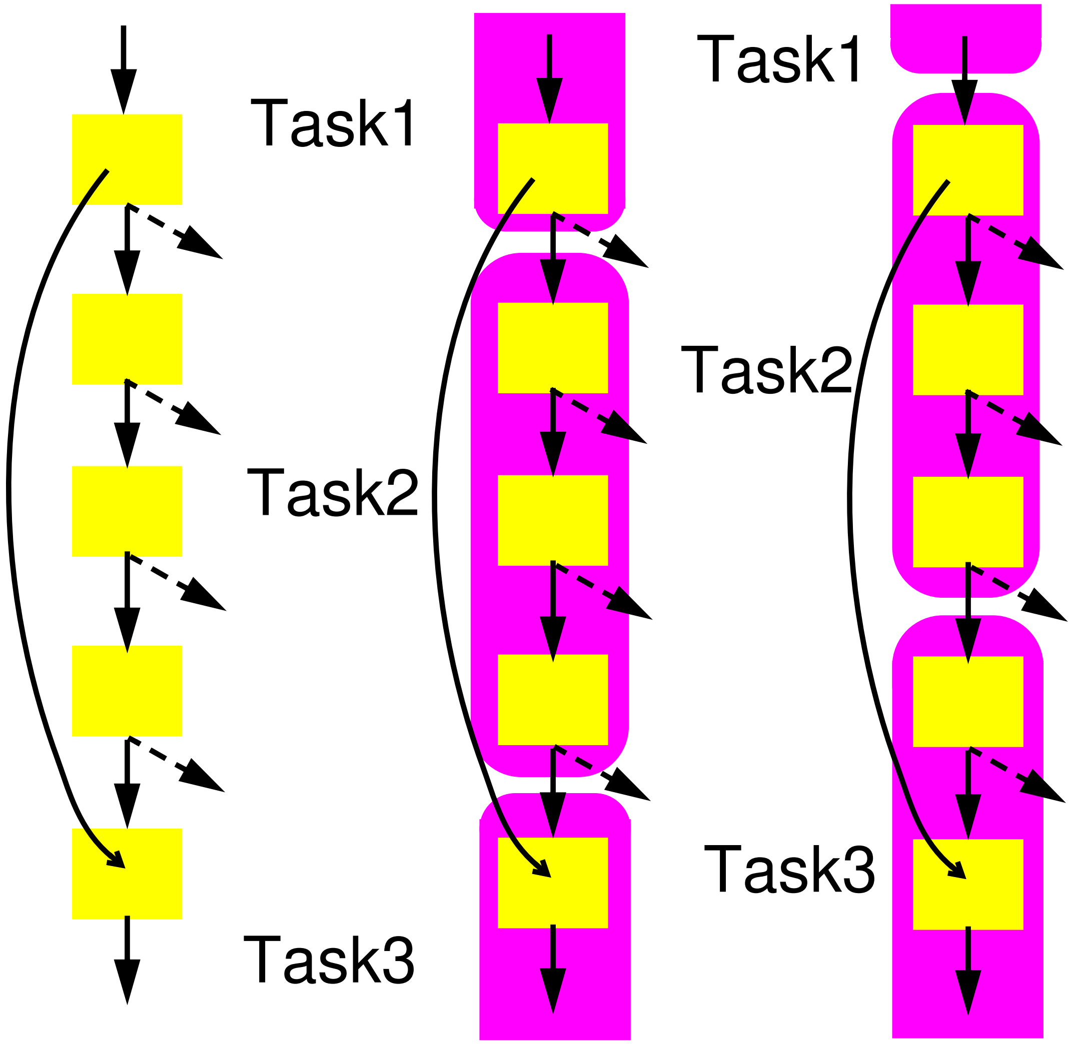
Schedule those that are “cut”

Dataflow analysis to identify the basic blocks to be included

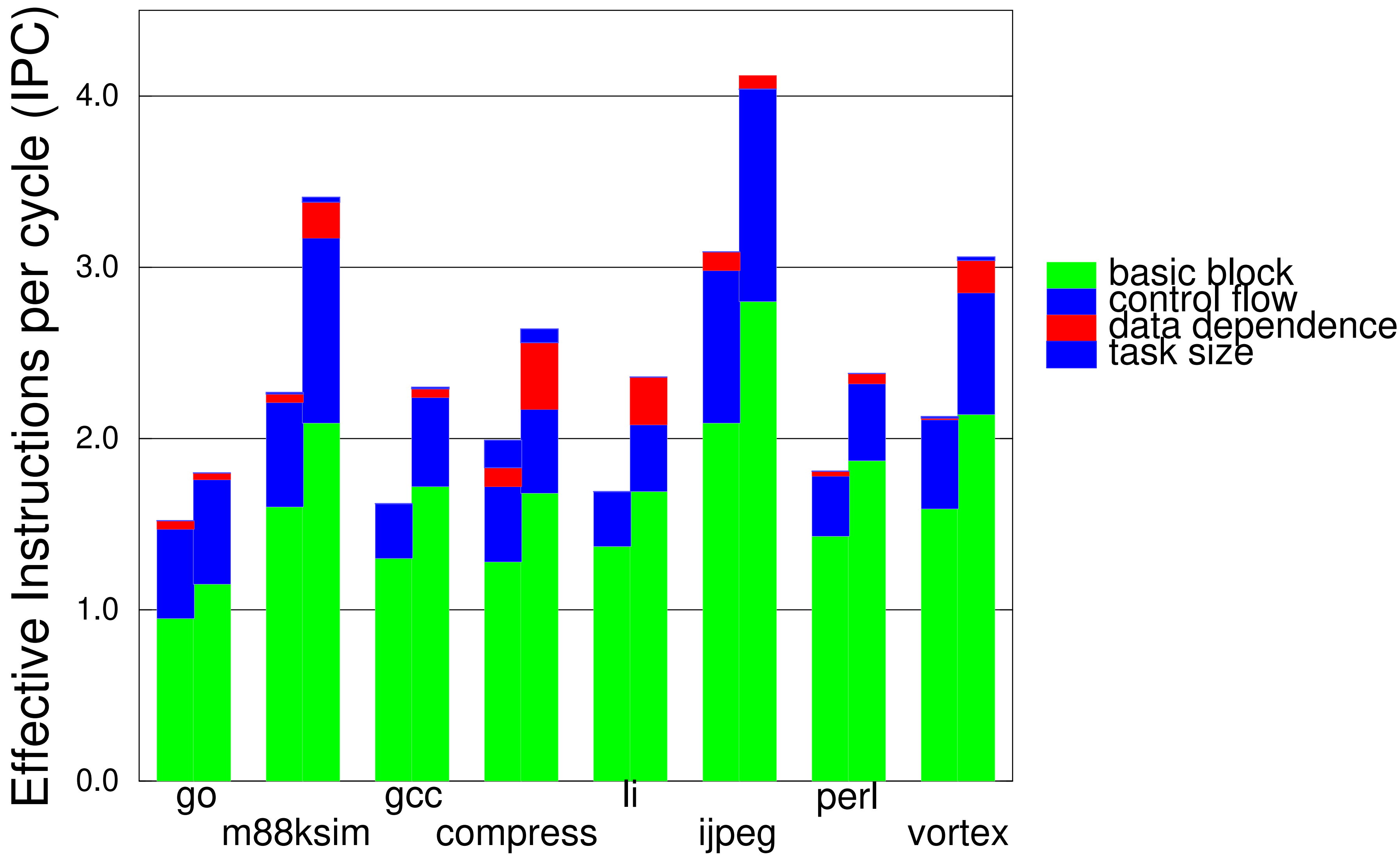
Heuristics: Data Dependence



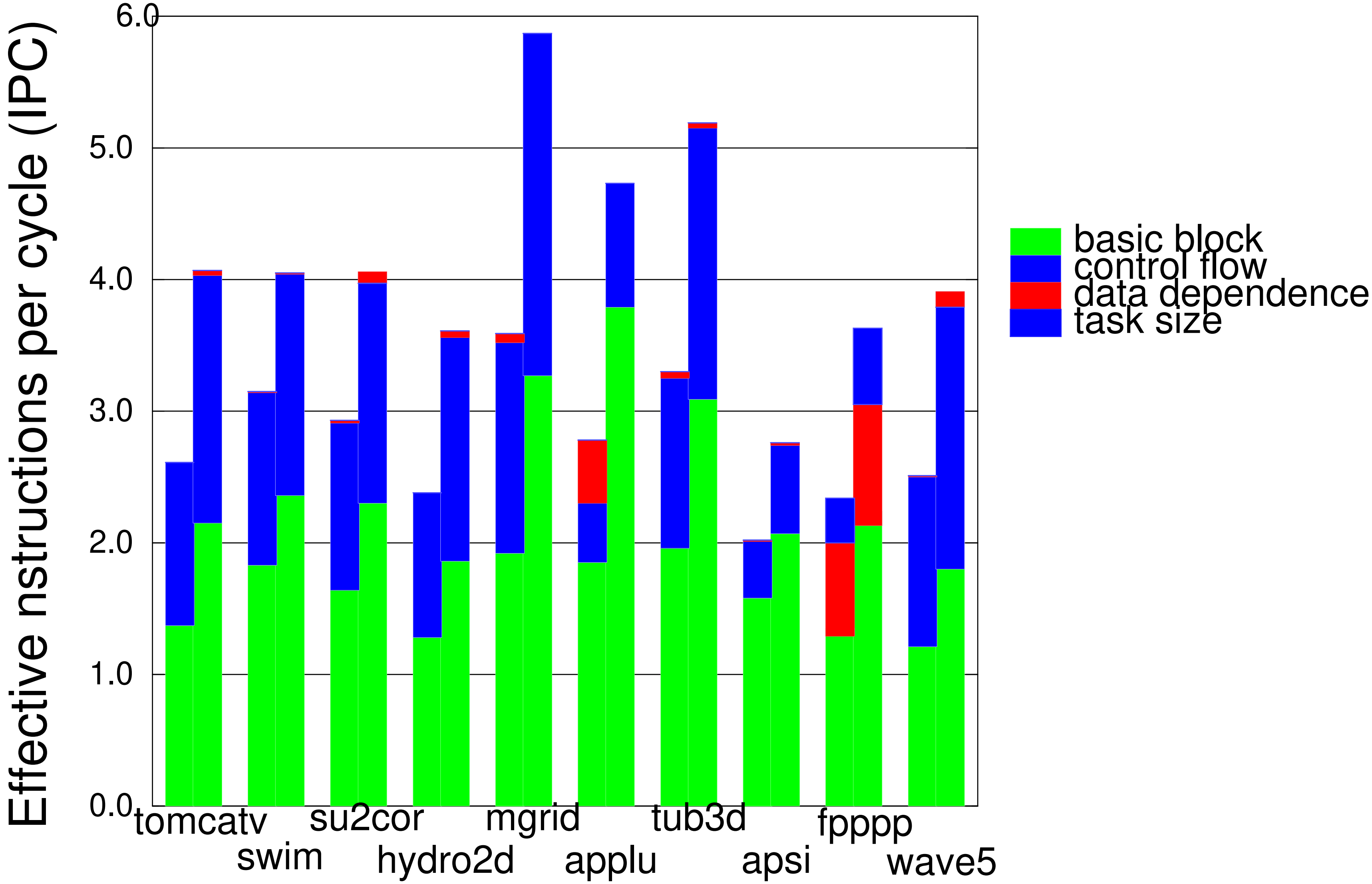
Heuristics: Data Dependence



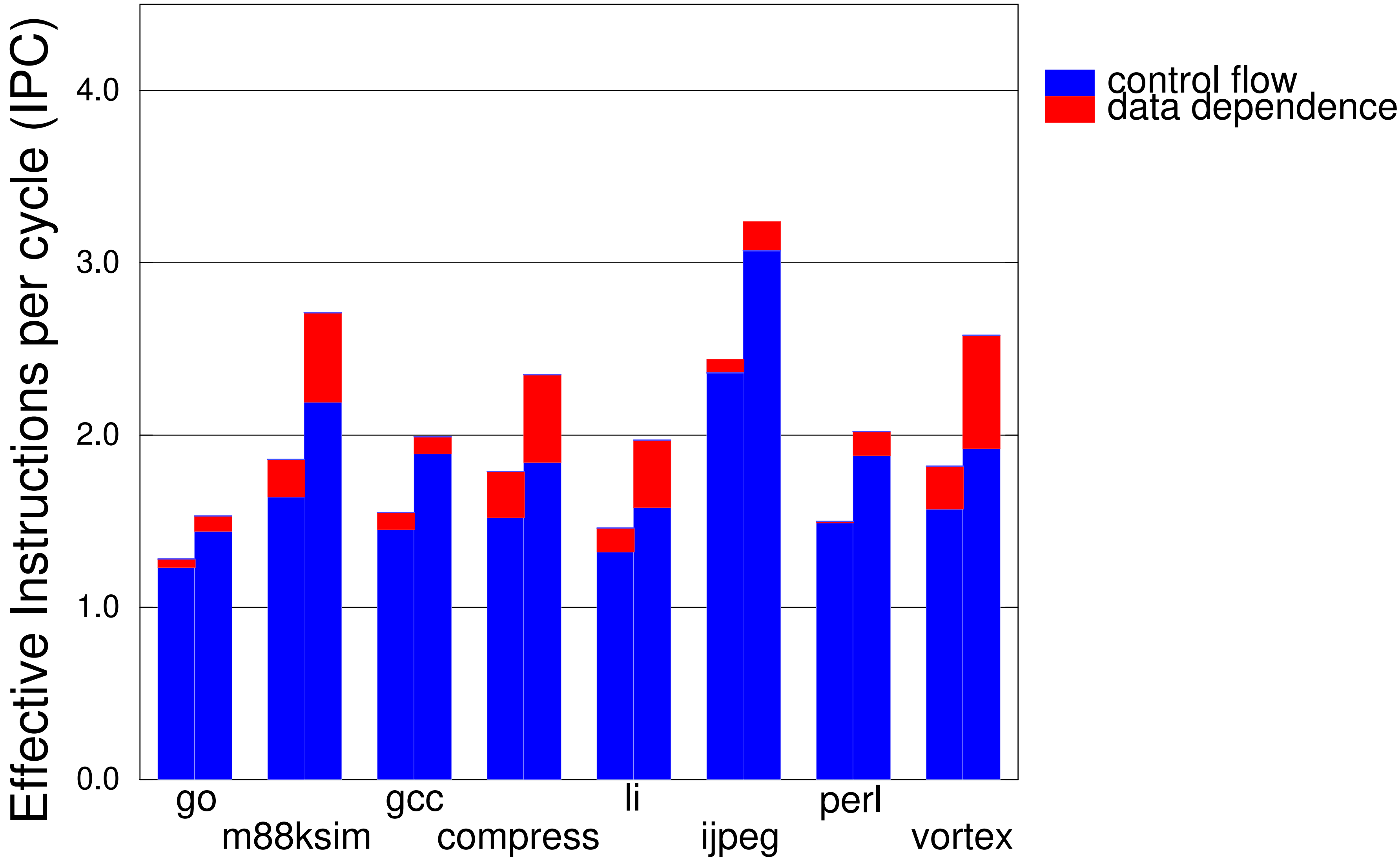
Task Selection Heuristics (OoO PUs)



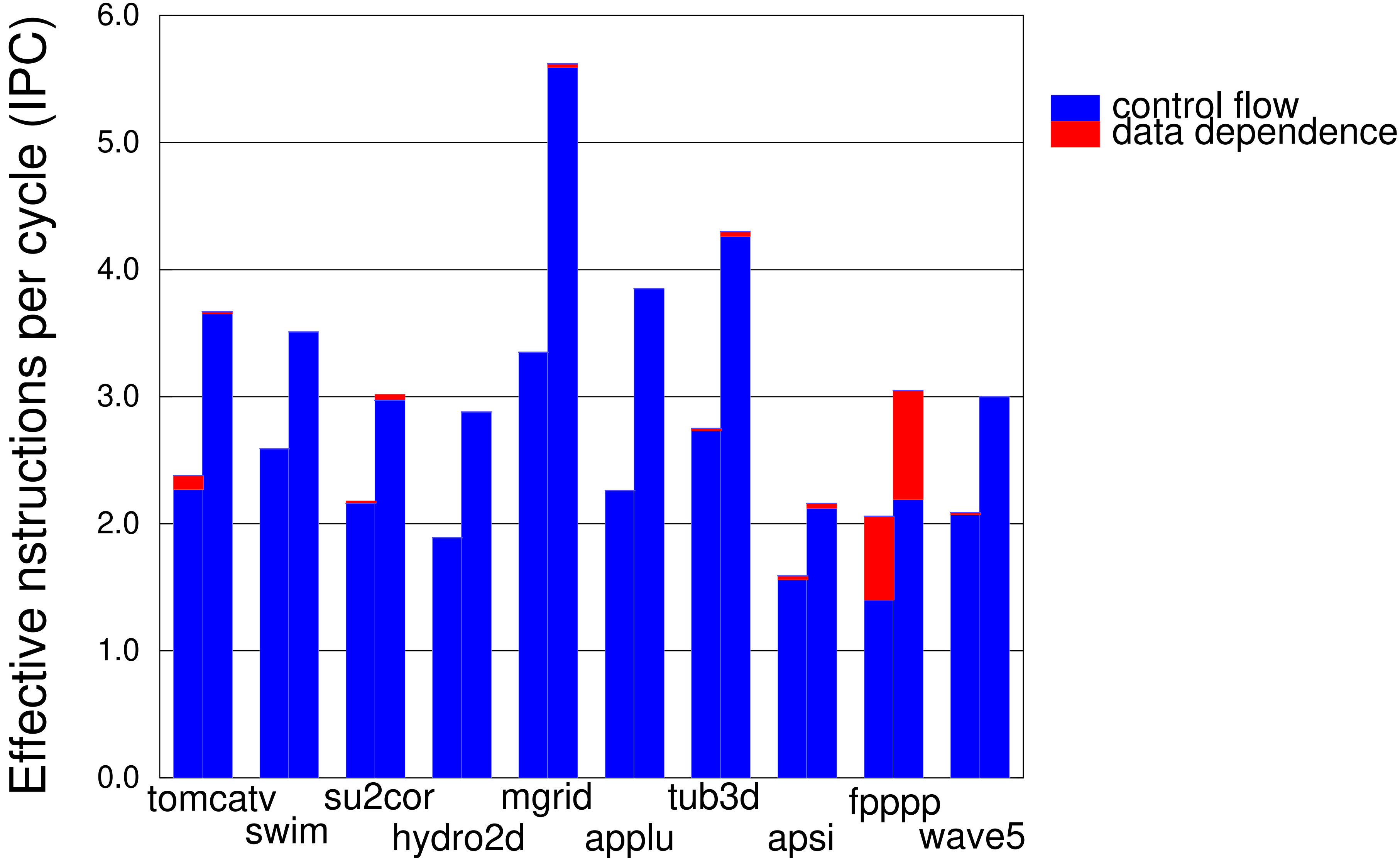
Task Selection Heuristics (OoO PUs)



Task Selection Heuristics (in-order PUs)



Task Selection Heuristics (in-order PUs)



Roadmap

~~Introduction~~

~~Task Selection~~

Inter-task register communication scheduling

Summary

Register Communication: Basics

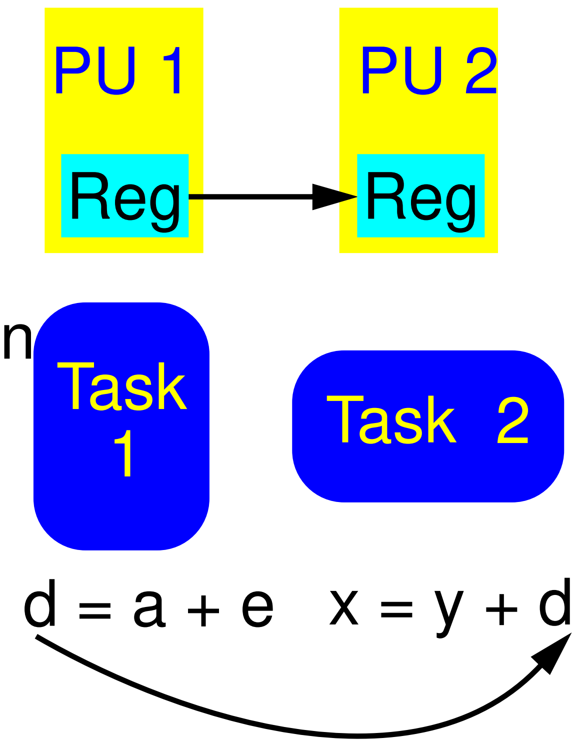
How do partitions honor original register dependencies?

Distributed register files but single register space

All registers are sent from one PU to next

1. Modified register sent after last modification

2. Unmodified registers sent as they arrive



How does the hardware know? Compiler tells it

Register Communication: Annotation

Which registers are modified?

- Create Mask: Registers that may be modified

When can they be sent?

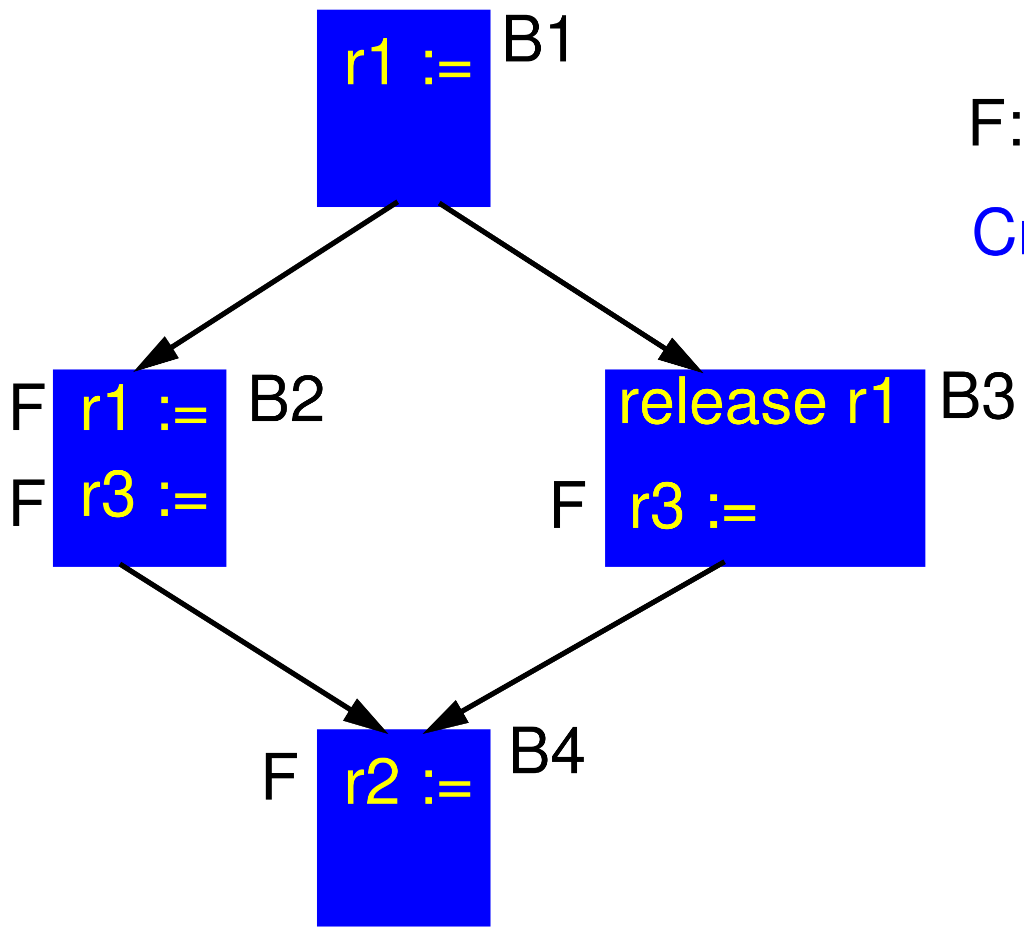
- Forward Bits: Send register values

Compiler uses dataflow analysis

Register communication is frequent

- Compiler places sends as early as possible

Register Communication: Example



F: Forward Bit on the instruction

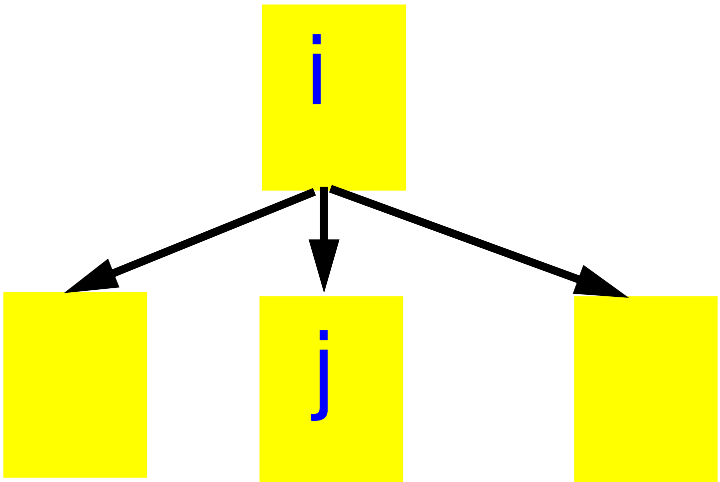
Create Mask: r1,r2,r3

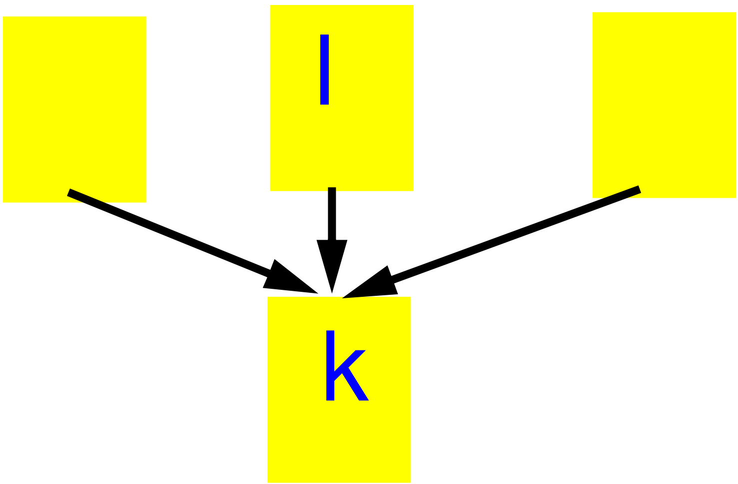
Register Communication: Some Details

$$NODEF(i, r) = \prod_{j \in children(i)} \{NODEF(j, r) \cap \neg BBDEF(j, r)\}$$

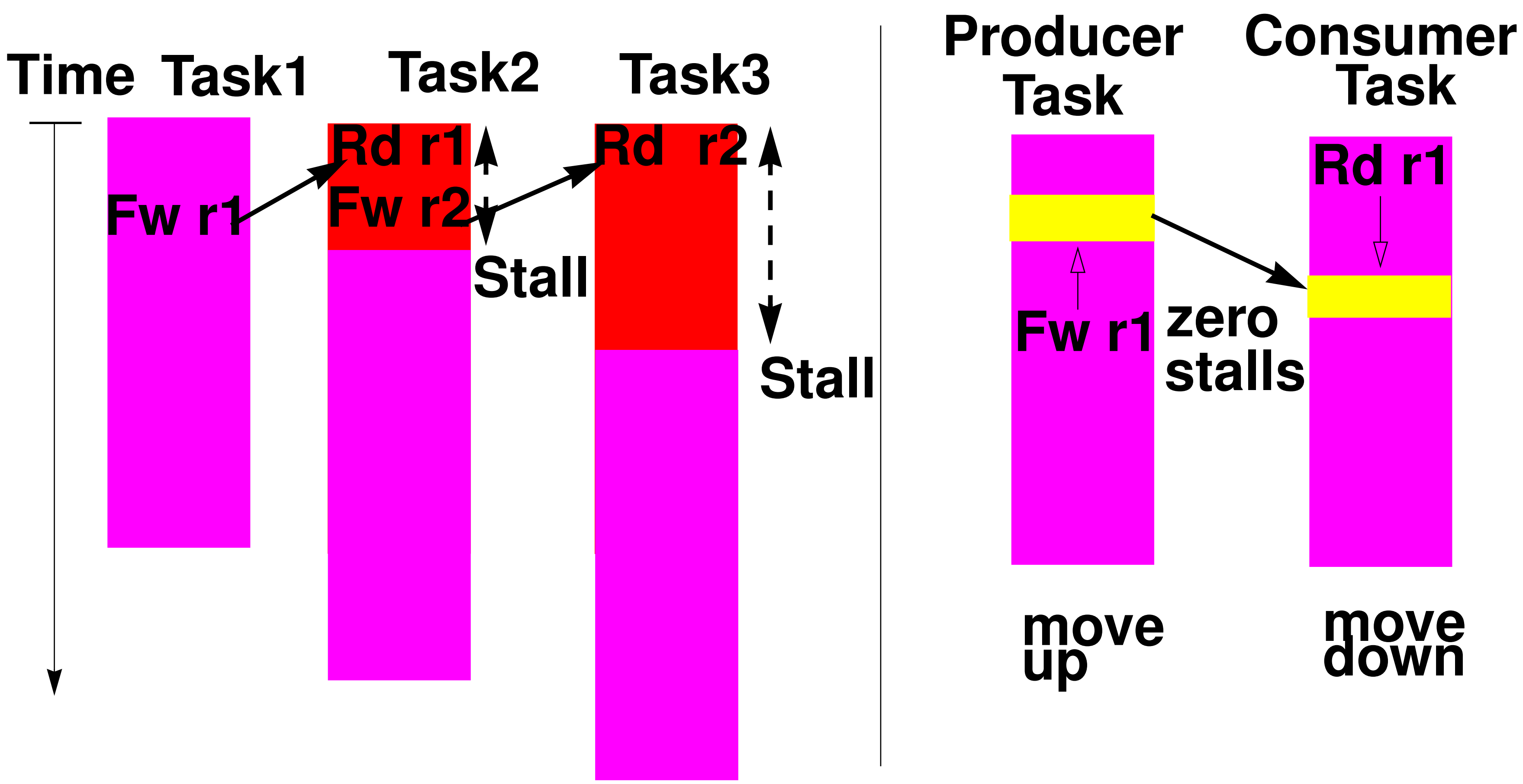
$$SEND(k, r) = \left\{ \sum_{l \in parents(k)} \neg NODEF(l, r) \right\} \cap NODEF(k, r)$$

Initial Values: $NODEF(i, r) = TRUE$





Register Communication Scheduling



Register Communication Scheduling

Identify the instructions to be moved

- Producers up, consumers down

Determine how much to be moved

- Cost model to estimate position

Perform code motion

- Move code across basic blocks using dataflow analyses

Cost Model

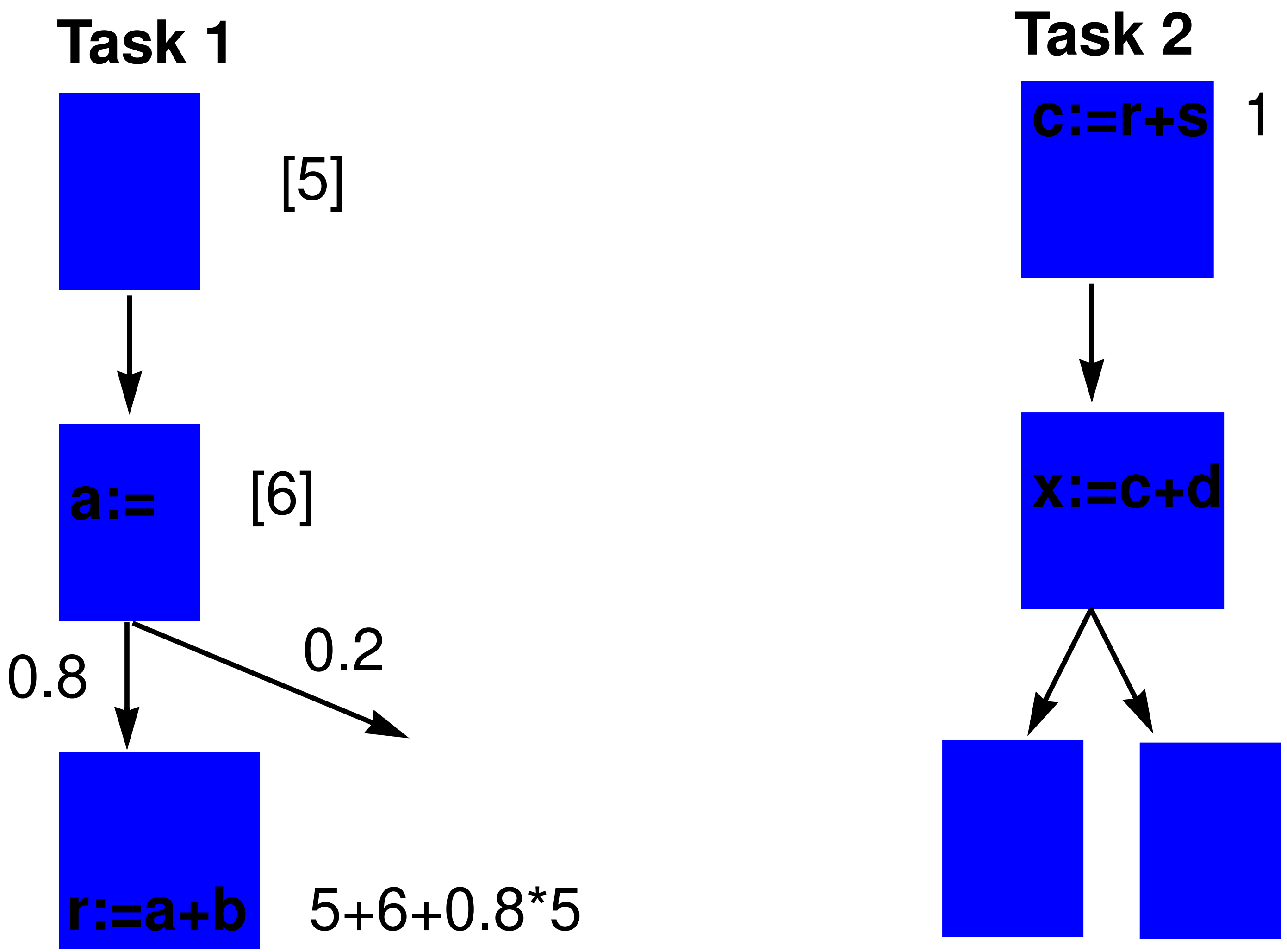
Estimate position by counting #instructions

Dynamic profiling used to obtain frequencies

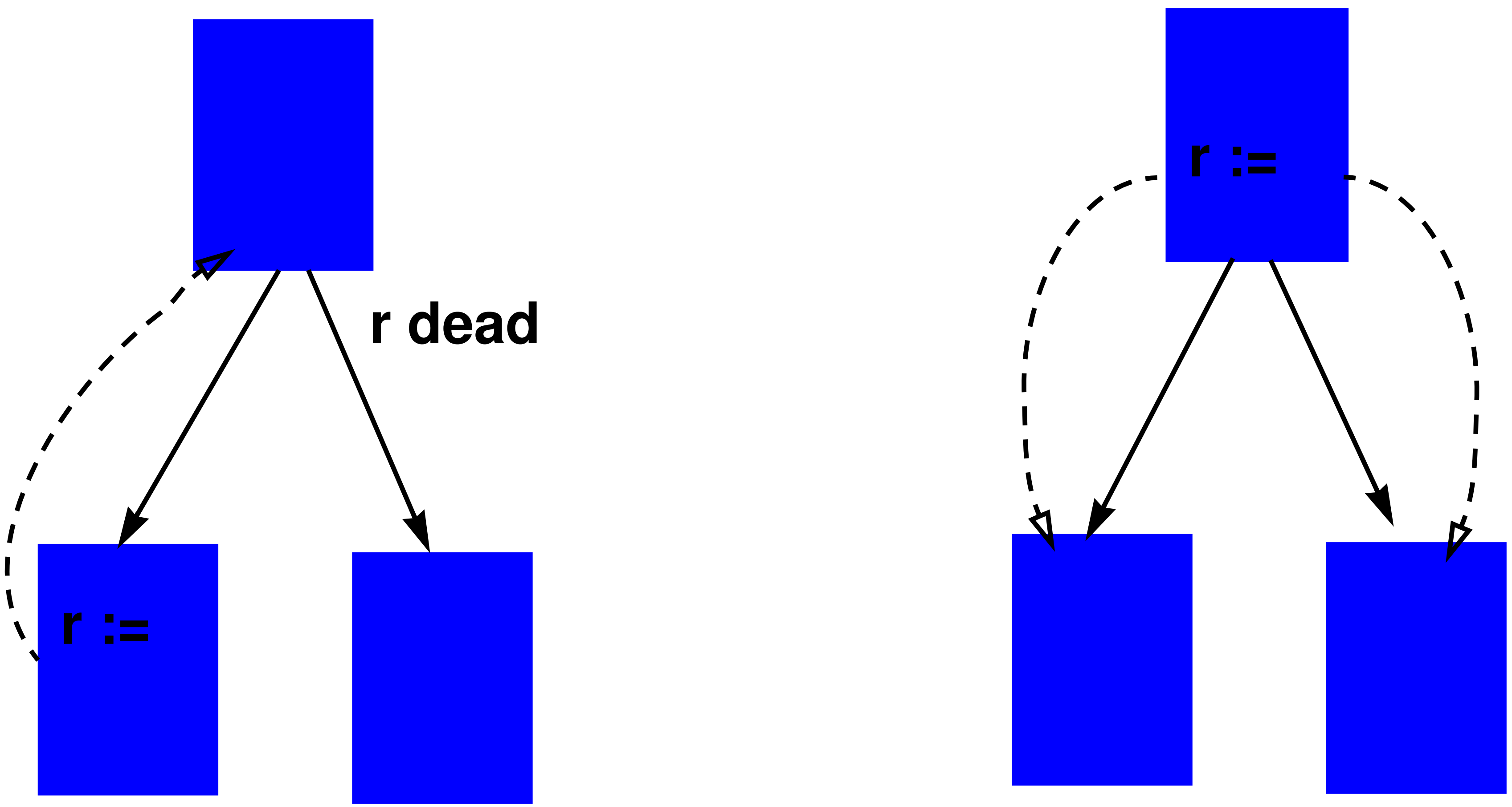
If loops are included in tasks, calculate position hierarchically

For function invocations use dynamic count

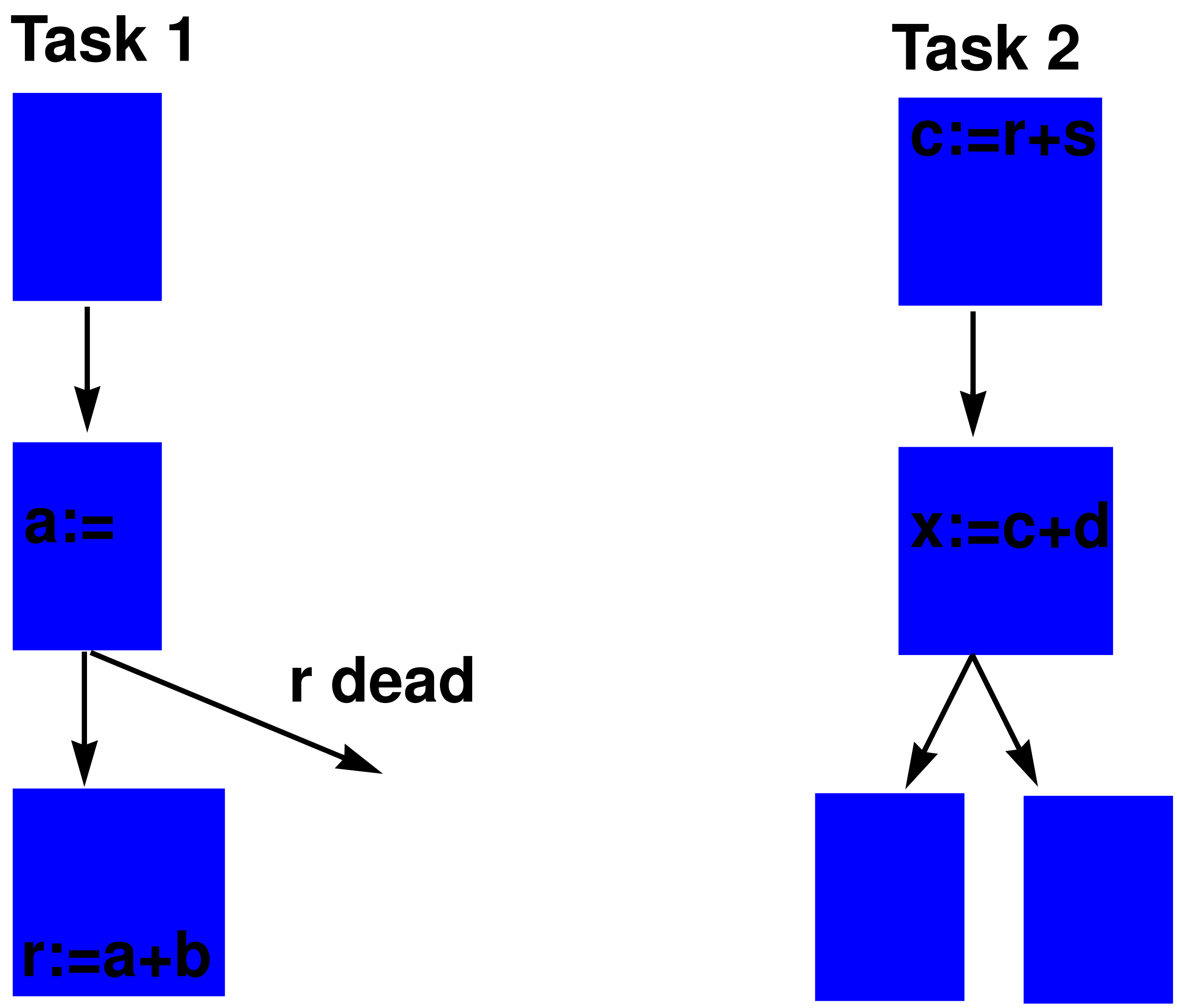
Cost Model



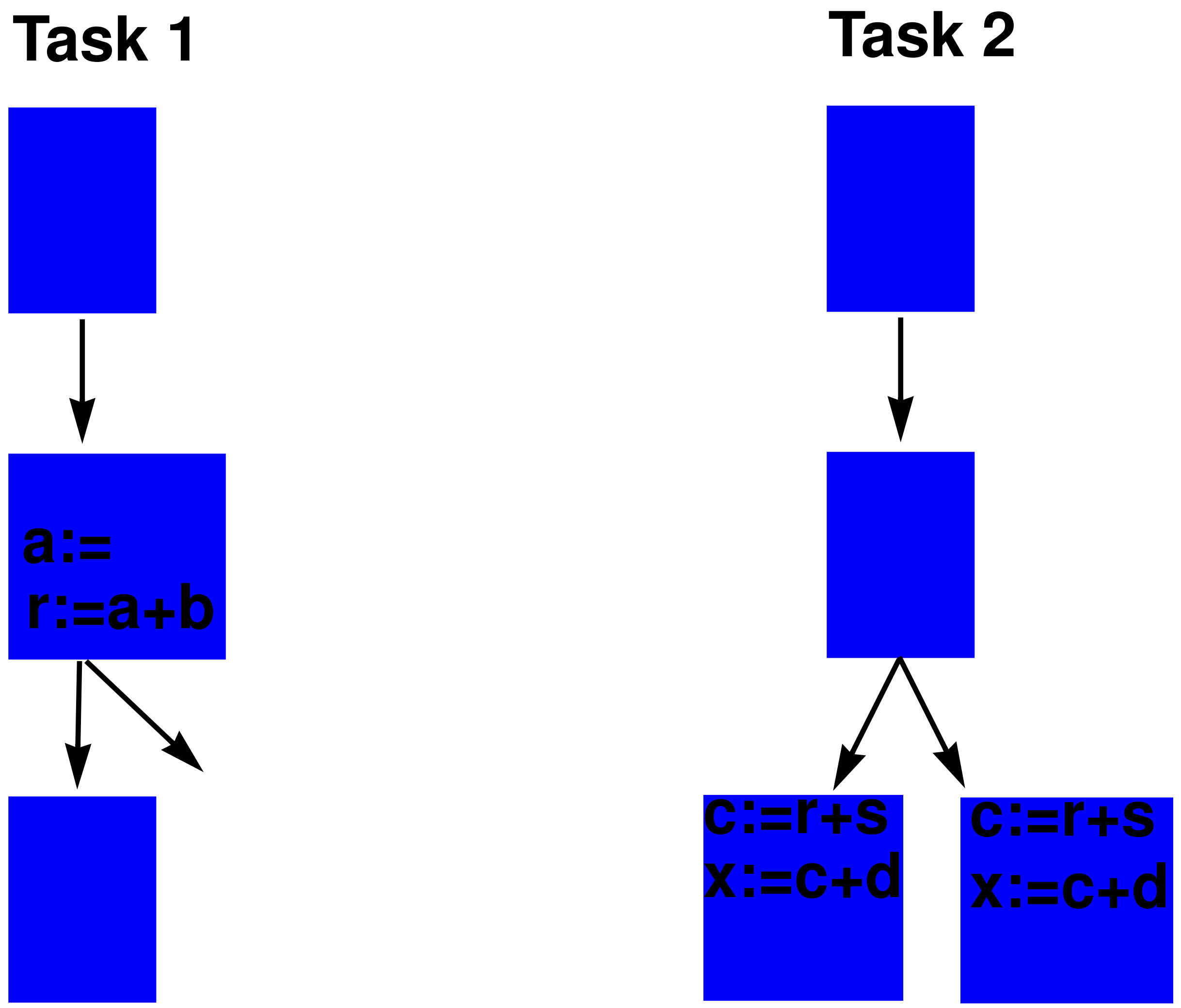
Code Motion



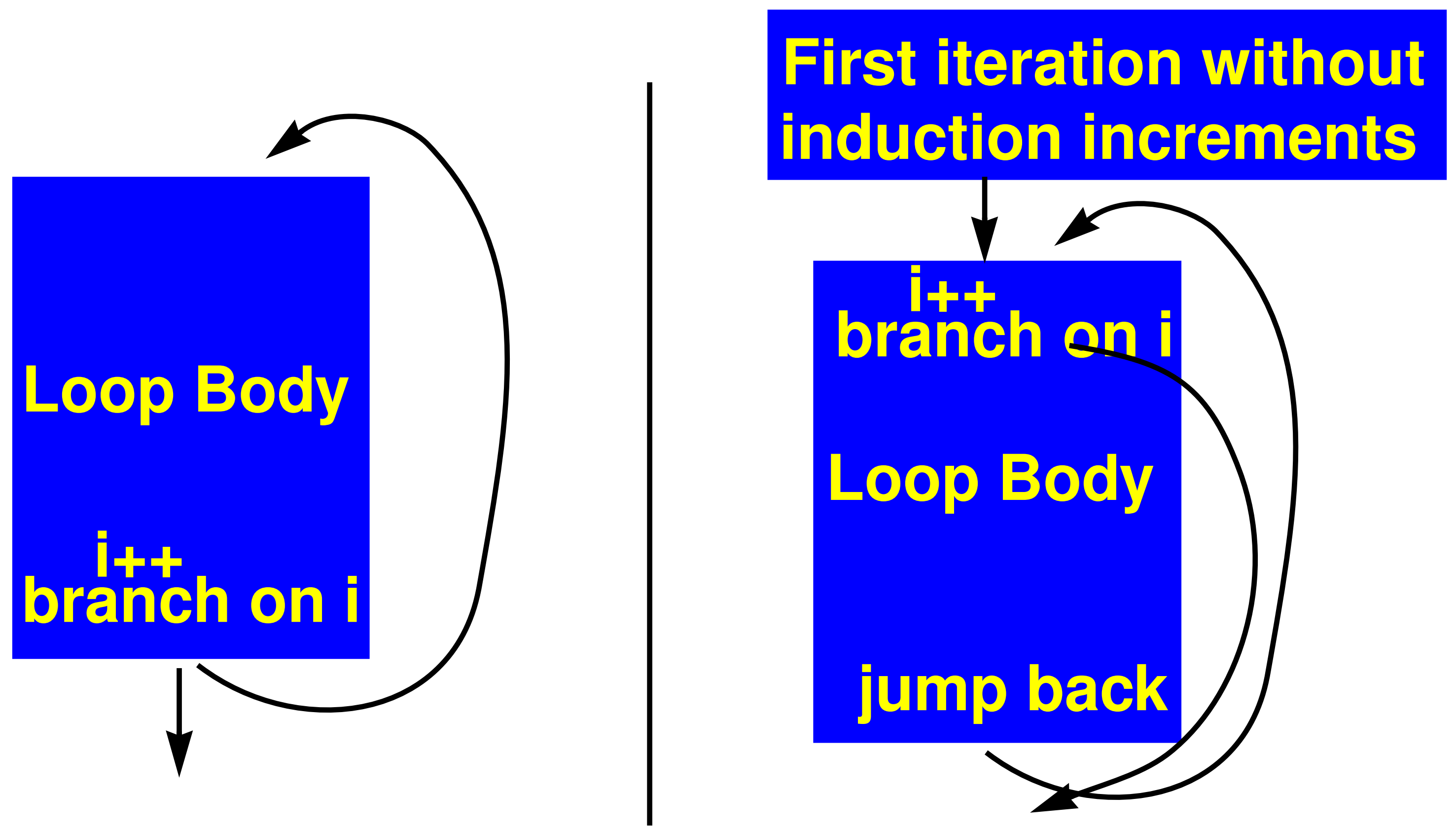
Scheduling Example



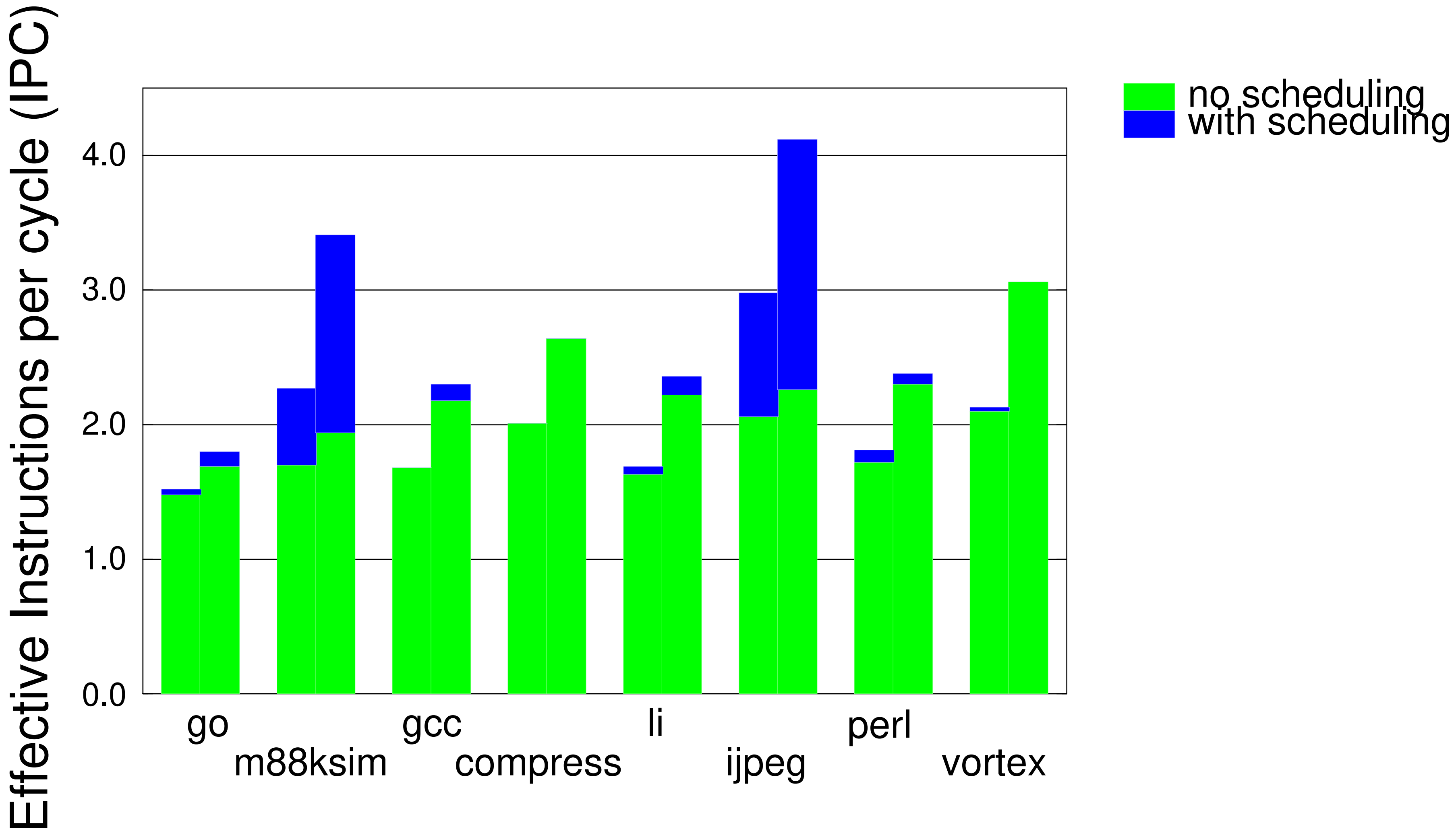
Scheduling Example



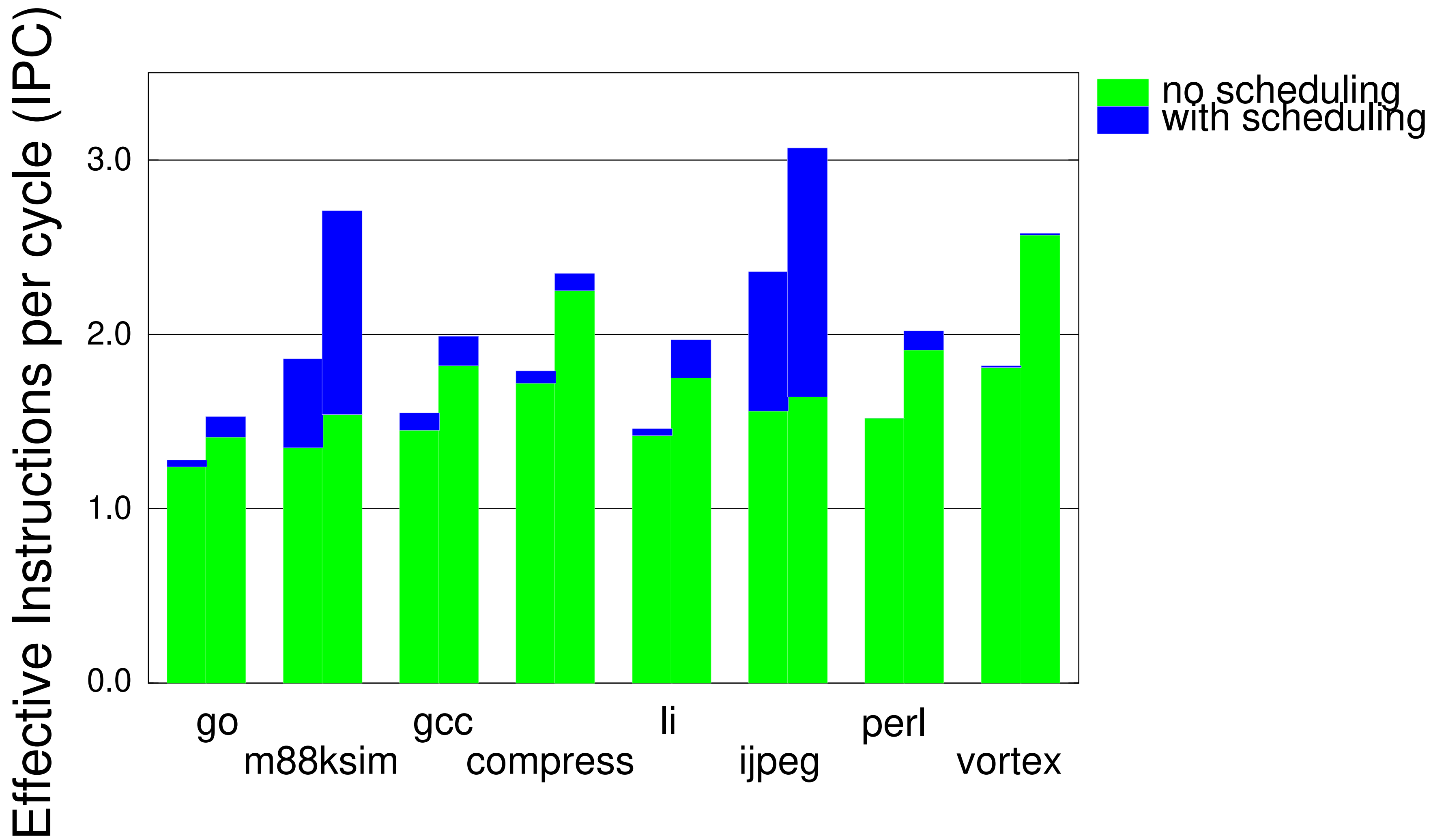
Loop Restructuring



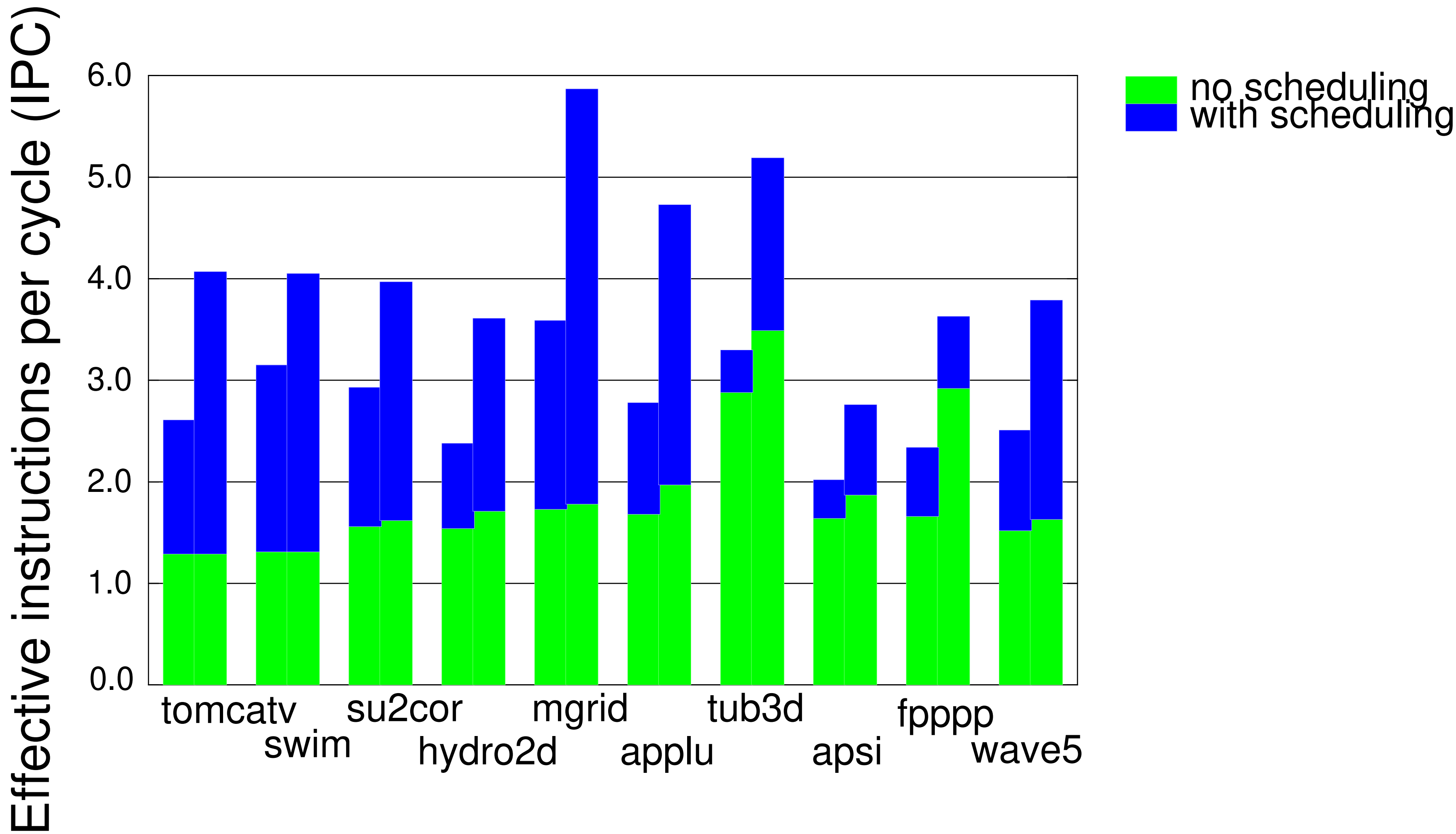
Register Communication Scheduling (OoO)



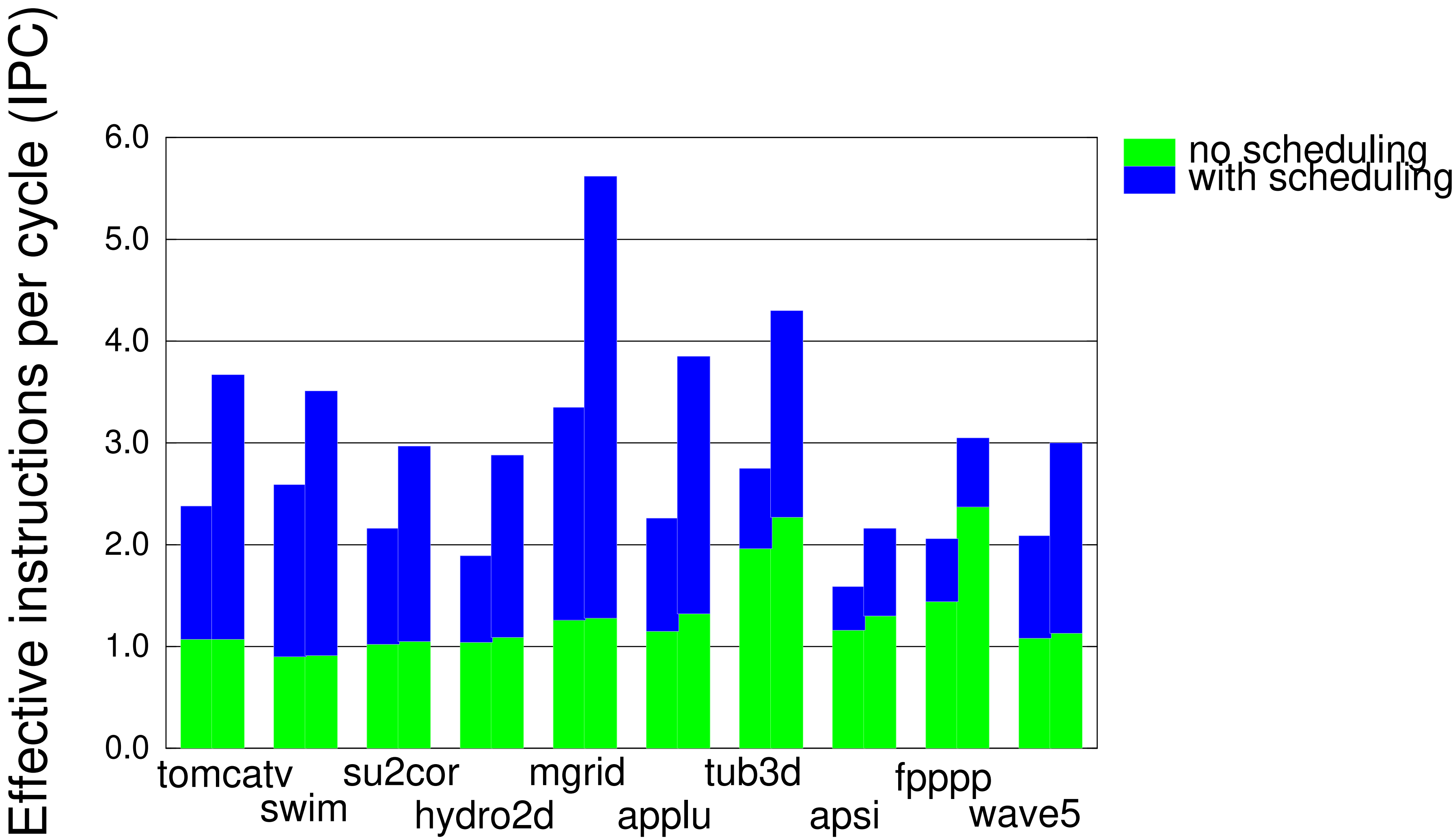
Register Communication Scheduling (i-o)



Register Communication Scheduling (OoO)



Register Communication Scheduling (i-o)



Task Size (SPEC95 INT)

Benchmarks	Basic Block	Control Flow		Data Dependence	
	total	control	total	control	total
099.go	6.4	2.53	18.2	2.04	12.7
124.m88ksim	4.3	2.97	14.8	2.42	10.3
126.gcc	5.8	2.52	12.4	2.32	11.6
129.compress*	5.7	1.78	10.2	2.77	15.0
130.li	3.9	1.89	8.1	1.64	7.1
132.ijpeg	10.6	2.42	23.3	2.43	23.8
134.perl	6.5	2.26	14.9	2.20	10.6
147.vortex	6.9	2.41	17.2	2.19	14.0

Task Size (SPEC95 FP)

Benchmarks	Basic Block	Control Flow		Data Dependence	
	total	control	total	control	total
101.tomcatv	44.1	4.96	114.9	3.24	84.8
102.swim	42.0	4.13	87.7	4.13	87.7
103.su2cor	49.8	8.03	107.8	8.03	107.8
104.hydro2d	11.9	6.0	44.0	5.24	39.5
107.mgrid	51.4	2.0	105.5	2.01	107.4
110.applu	21.7	1.71	39.0	1.70	38.5
125.turb3d	21.2	2.46	41.7	2.44	40.8
141.apsi	24.8	2.78	51.0	2.63	46.8
145.fpppp*	957.8	1.45	59.0	2.50	66.5
145.wave5	24.4	4.20	59.1	4.08	56.1

Control Flow Misspeculation Rate (INT)

Benchmarks	Basic Block	Control Flow		Data Dependence	
		Task	Branch	Task	Branch
099.go	14.4%	14.7%	5.8%	14.6%	7.2%
124.m88ksim	3.1%	4.0%	1.4%	4.9%	2.0%
126.gcc	4.4%	5.8%	2.3%	7.4%	3.2%
129.compress	5.0%	5.7% *	3.2% *	7.8% *	2.8% *
130.li	3.3%	4.0%	2.1%	5.2%	3.2%
132.ijpeg	6.0%	3.7%	1.5%	5.1%	2.1%
134.perl	1.6%	3.9%	1.7%	4.7%	2.1%
147.vortex	0.8%	0.7%	0.3%	0.7%	0.3%

Control Flow Misspeculation Rate (FP)

Benchmarks	Basic Block	Control Flow		Data Dependence	
		Task	Branch	Task	Branch
101.tomcatv	1.6%	0.4%	0.1%	0.4%	0.1%
102.swim	0.1%	0.2%	0.0%	0.2%	0.0%
103.su2cor	3.4%	0.5%	0.1%	0.5%	0.1%
104.hydro2d	0.1%	0.3%	0.1%	0.2%	0.0%
107.mgrid	1.1%	2.2%	1.1%	2.2%	1.1%
110.applu	3.9%	3.9%	2.3%	4.2%	2.5%
125.turb3d	3.4%	5.8%	2.4%	6.7%	2.7%
141.apsi	2.9%	4.3%	1.5%	4.1%	1.6%
145.fpppp	5.6%	1.8% *	1.2% *	2.8% *	1.1% *
145.wave5	0.8%	0.8%	0.2%	1.1%	0.3%

Roadmap

~~Introduction~~

~~Task Selection~~

~~Inter-task register communication scheduling~~

Summary

Summary

Splits big window and wide machine across several little PUs

Important performance issues:

- Inter-task control flow
- Inter-task data dependences (register)
- Task overheads
- Load imbalance

Task selection is affects many performance issues

Summary

I constructed a compiler to study program architecture interaction

- Partition a sequential program into tasks
- Generate inter-task register communication
- Schedule inter-task register communication
- Loop restructuring and dead register optimizations
- Provide program information

Task selection heuristics better utilize hardware

Scheduling improves performance modestly to significantly

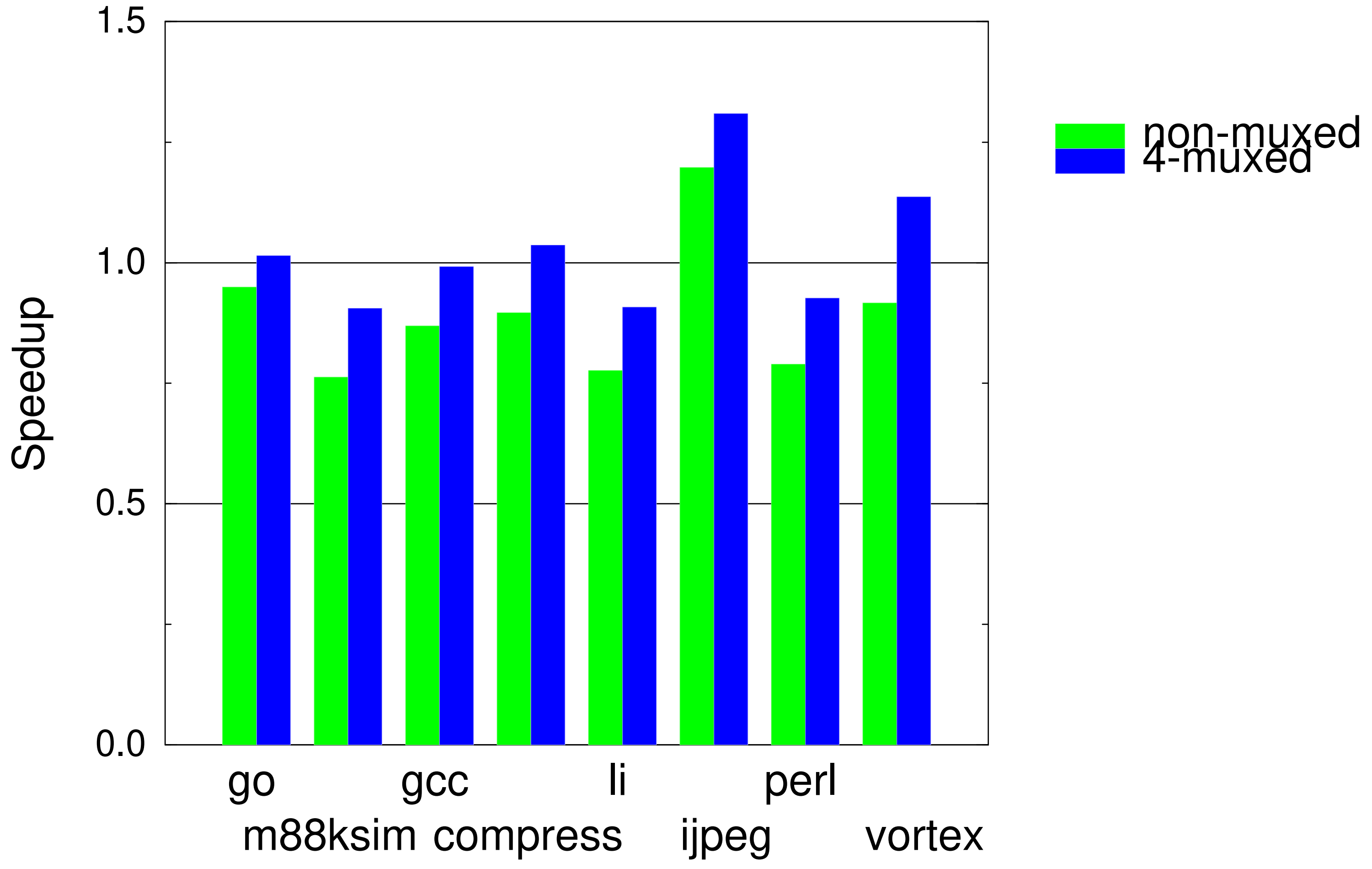
Both are more important for larger number of PUs

Comparison with Multiprocessors

Attributes	Multiprocessor	Multiscalar
Speculative task initiation	No/Difficult	Yes
Multiple flows of control	Yes	Yes
Task determination	Static	Static (possibly dynamic)
Software guarantee of inter-task control independence	Required	Not required
Software knowledge of inter-task data dependences	Required	Not required
Inter-task sync.	Explicit	Implicit/Explicit
Inter-task communication	Through memory Through messages	Through registers and memory
Register space	Distinct for PEs	Common for PEs
Memory space	Common Distinct	Common for PEs

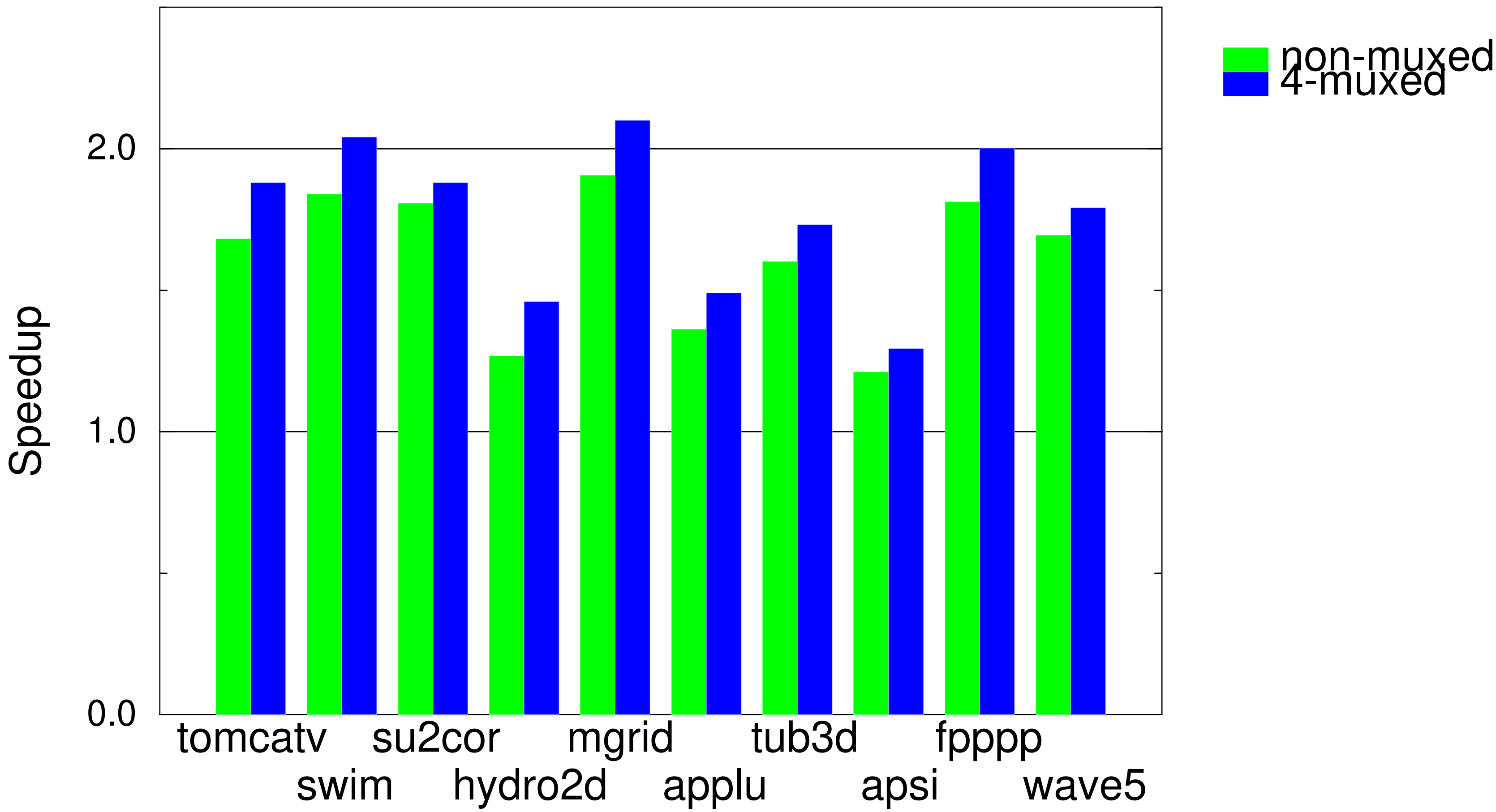
Superscalar vs. Multiscalar(8-way vs 4x2)

EQUAL CLOCK CYCLE



Superscalar vs. Multiscalar(8-way vs 4x2)

EQUAL CLOCK CYCLE



Why compiler

Interaction between programs and architecture

Response to program transformations

Flexibility and cost of analyses

Within the limitations of a real compiler implementation