

Register Allocation by Puzzle Solving

Fernando M Q Pereira

Jens Palsberg

UCLA

University of California, Los Angeles

Presented at PLDI 2008

Call for papers and participation

- **Jul 15, 2009: POPL submission deadline**
 - **Principles of Programming Languages**
 - **POPL will be in Madrid, Jan 20-22, 2010**
- **Aug 9-15, 2009 at UCLA:**
 - **LICS, IEEE Logic in Computer Science**
 - **SAS, Static Analysis Symposium**
 - **Six workshops**

Three messages

- **SSA-based register allocation for x86**
- **Register allocation = solving puzzles**
- **Split live ranges everywhere!**

A compiler

source language



parser

intermediate representation

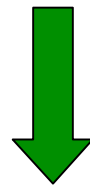


code generator

machine code

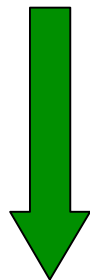
A better compiler

source language



parser

intermediate representation



code generator with a

register allocator

machine code

What is register allocation?

A = 10

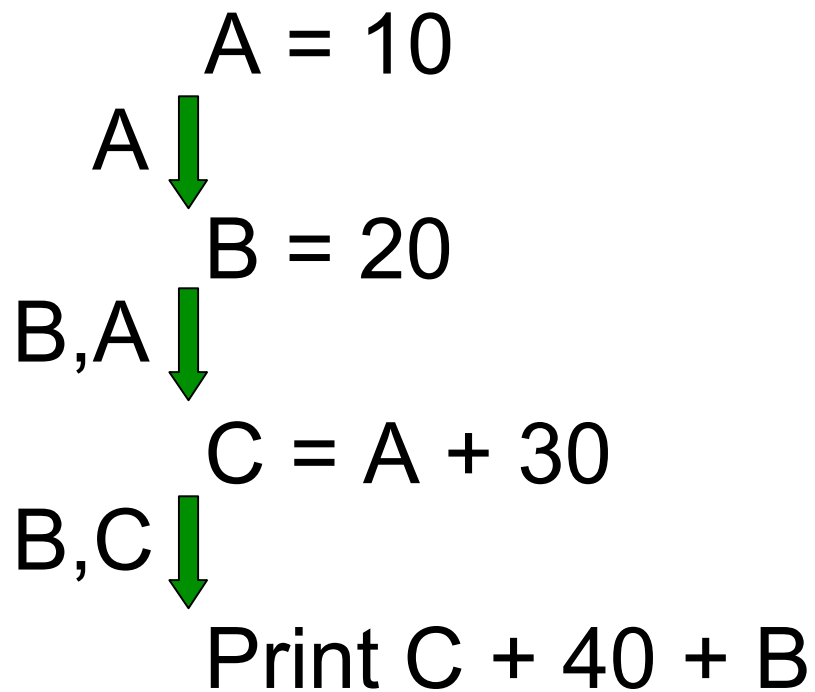
B = 20

C = A + 30

Print C + 40 + B

**Assume we have
two registers**

Reg. allocation = liveness analysis + graph coloring



Interference graph:

A — B — C

With colors:

A — B — C

After register allocation

A = 10

B = 20

C = A + 30

Print C + 40 + B

A — B — C

R1 = 10

R2 = 20

R1 = **R1** + 30

Print **R1** + 40 + **R2**

Many models of register allocation

- **Graph coloring**
 - **George & Appel TOPLAS '96, iterated register coalescing**
 - **Smith, Ramsey & Holloway, PLDI '04, aliased registers**
- **Linear scan**
 - **Poletto & Sarkar TOPLAS '99, excellent for JIT compilers**
- **Integer linear programming**
 - **Appel & George PLDI '01, optimal spilling**
- **Partitioned Boolean quadratic programming**
 - **Scholz & Eckstein SCOPES '02, optimal spilling**
- **Multi-commodity network flow**
 - **Koes & Goldstein PLDI '06, iterative within a time budget**

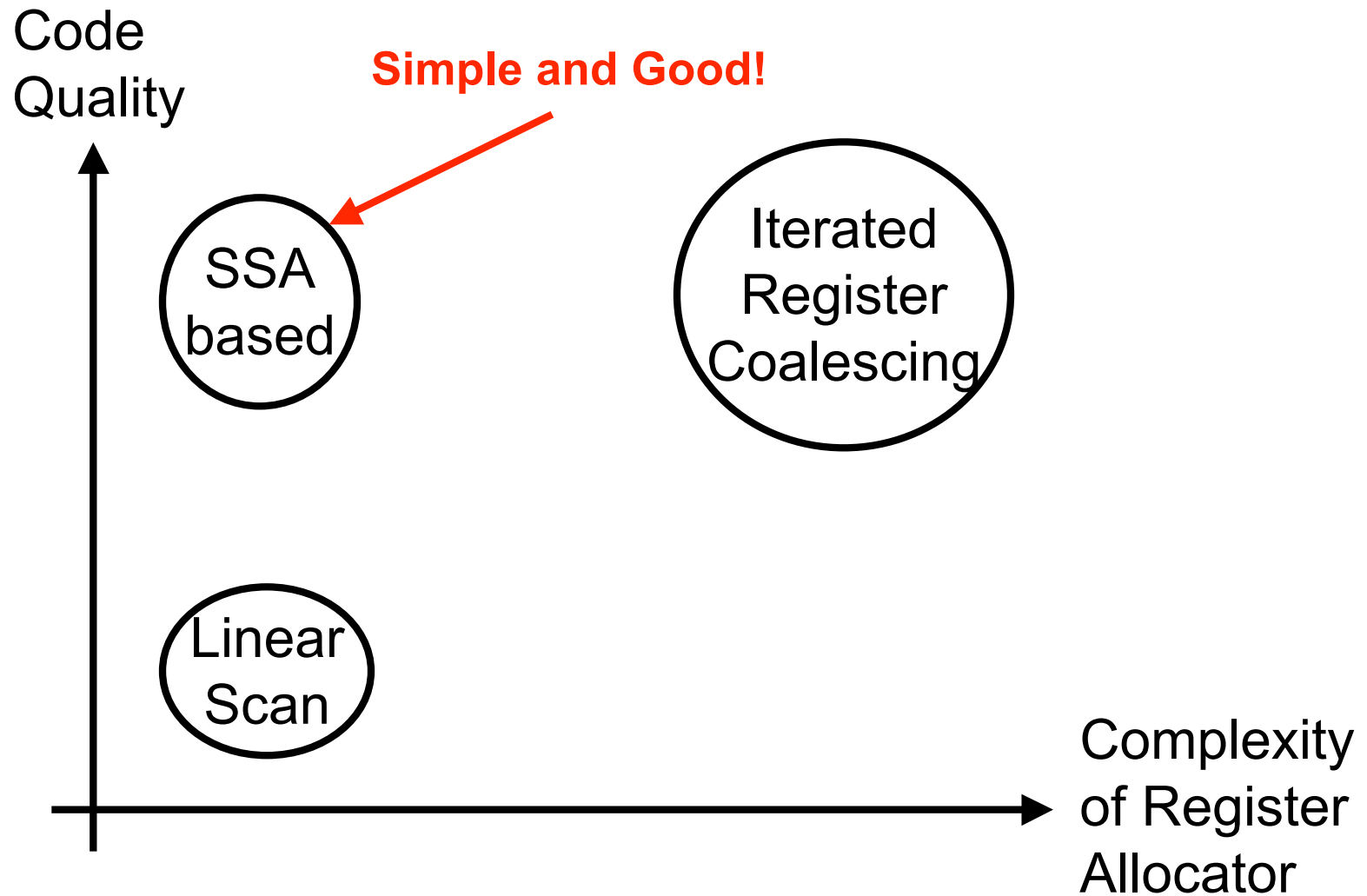
Complications

- **Spill**: If there are not enough registers, then represent the remaining variables in memory
- **Coalesce**: For an assignment $x=y$, try to map x,y to the same register
- **Live-range splitting**: Enable a variable to some times be in a register and some times in memory
- **Pre-coloring**: some instructions require it
- **Register aliasing and pairing**: found on x86, ARM, SPARC V8+V9, etc
- **Rematerialization**: don't store, recompute!

SSA to the rescue

- **Theorem: a program in strict SSA form has a chordal interference graph**
 - **Enables two-phase register allocation:**
 - 1. Spilling**
 - 2. Coloring and coalescing**
- Goal of spilling: make**
- $$\text{need-for-registers} \leq \# \text{ registers}$$
- **If all the registers have the same size, then**
- $$\text{need-for-registers} = \text{size}(\text{largest clique})$$

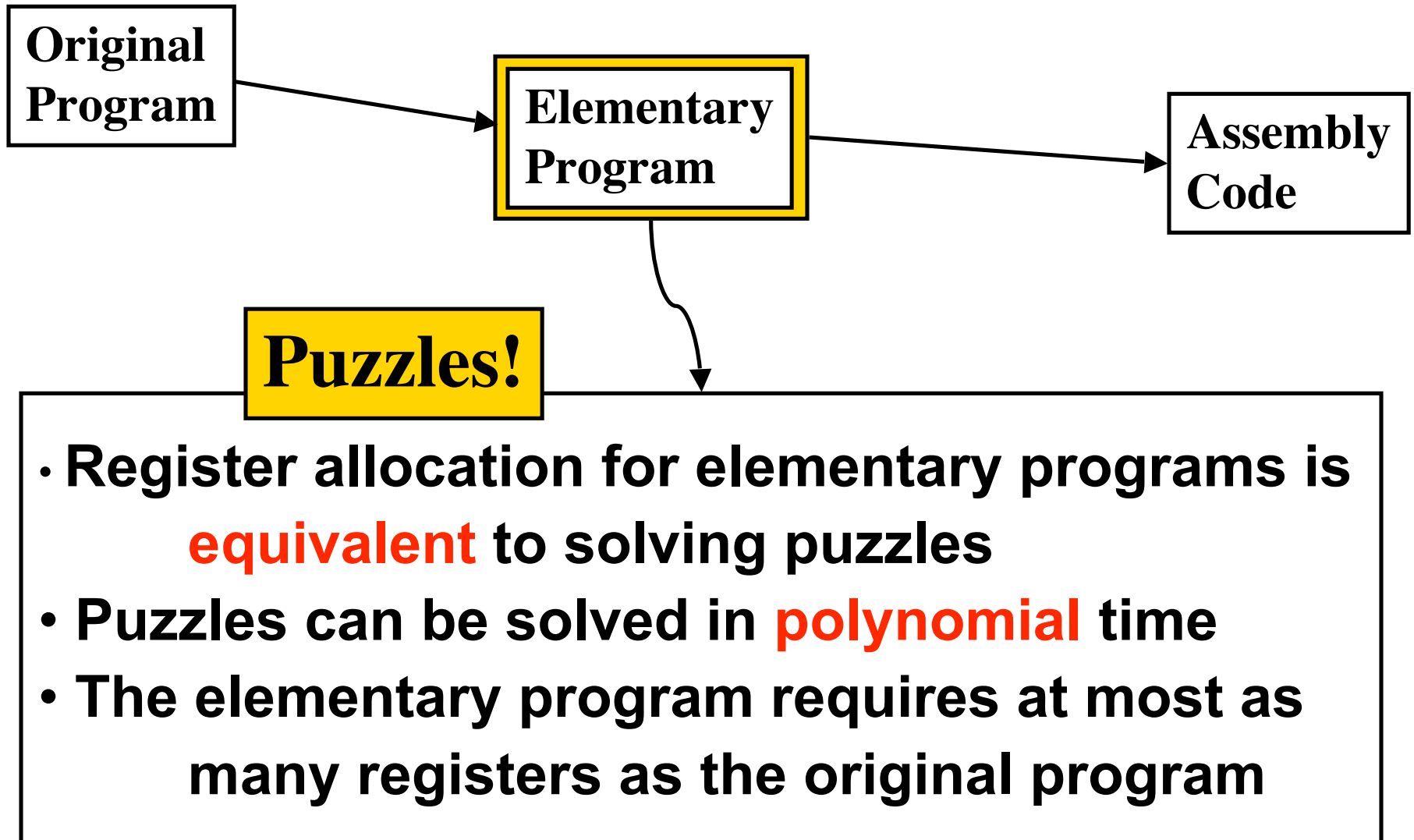
Comparison



Our goal

- **Our goal: implement two-phase register allocation for **x86** and handle pre-coloring, and register aliasing and pairing**
- **Problem 1: with pre-coloring, computing **need-for-registers** is **NP-complete for unit interval graphs** [Marx, 2006]**
- **Problem 2: with register aliasing and pairing, computing **need-for-registers** is **NP-complete for interval graphs** [Lee, Palsberg, Pereira, ICALP 2007]**

Puzzles: a new approach to register allocation

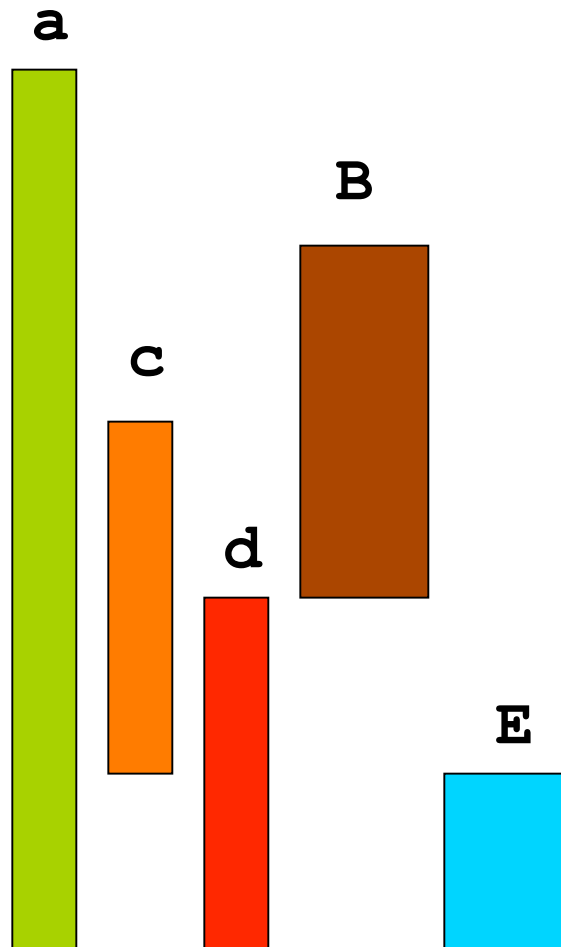


Example

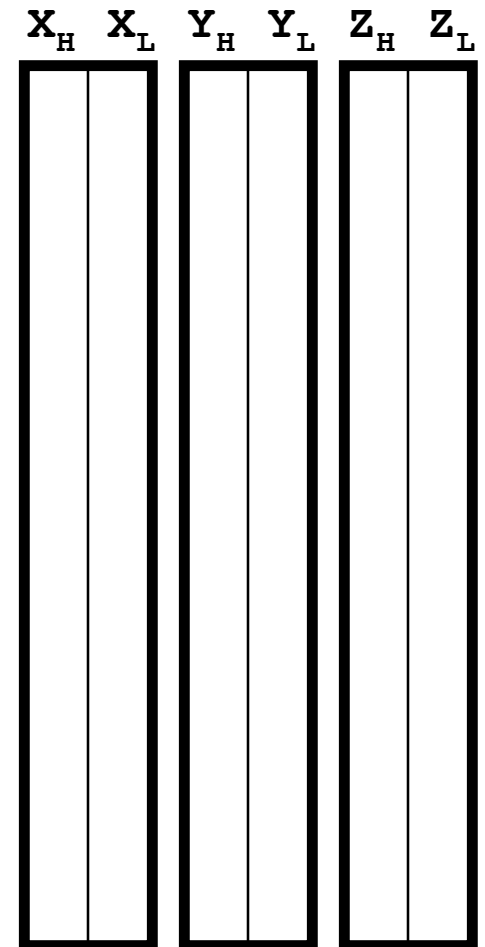
Program

$a = \bullet$
 $B = \bullet$
 $c = \bullet$
 $d = B$
 $E = c$
 $\bullet = a, d, E$

Live Ranges



Registers



An optimal solution

Program

$X_H = \bullet$

$Y = \bullet$

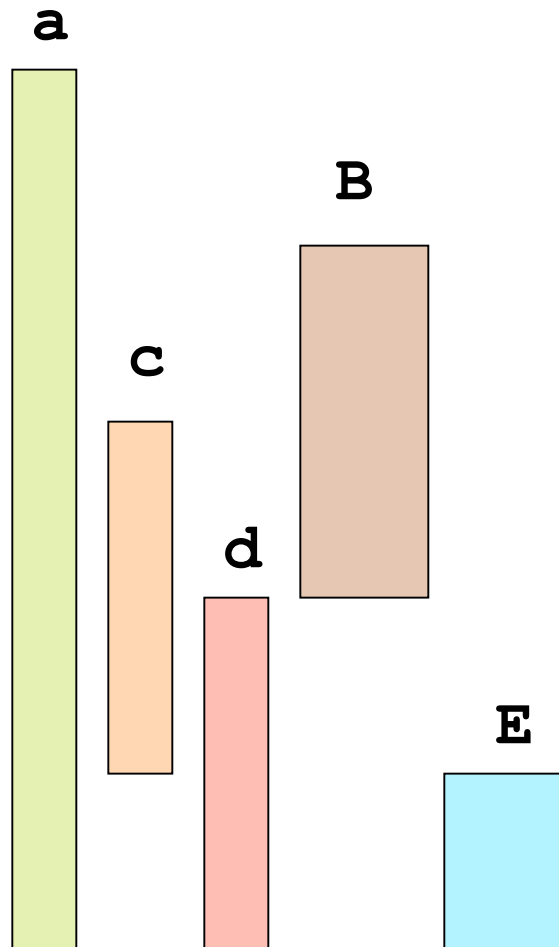
$X_L = \bullet$

$Y_H = Y$

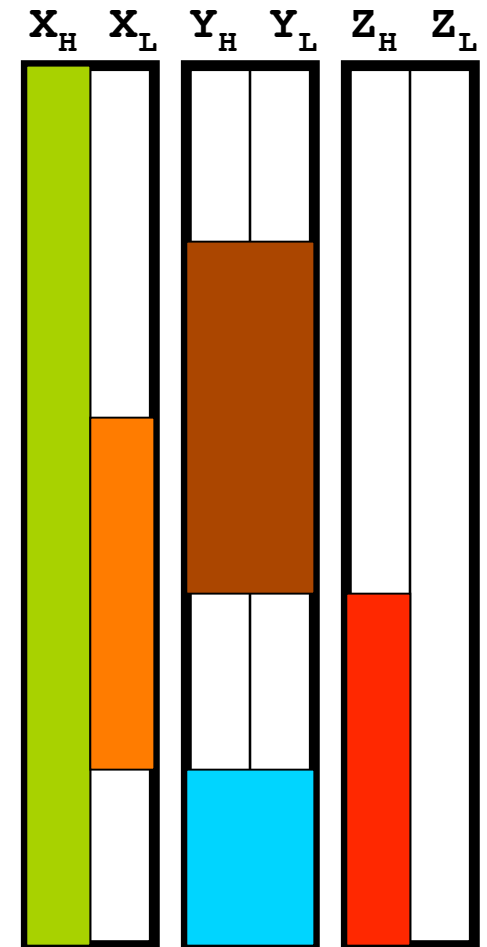
$Z = X_L$

$\bullet = X_H, Y_H, Z$

Live Ranges



Registers

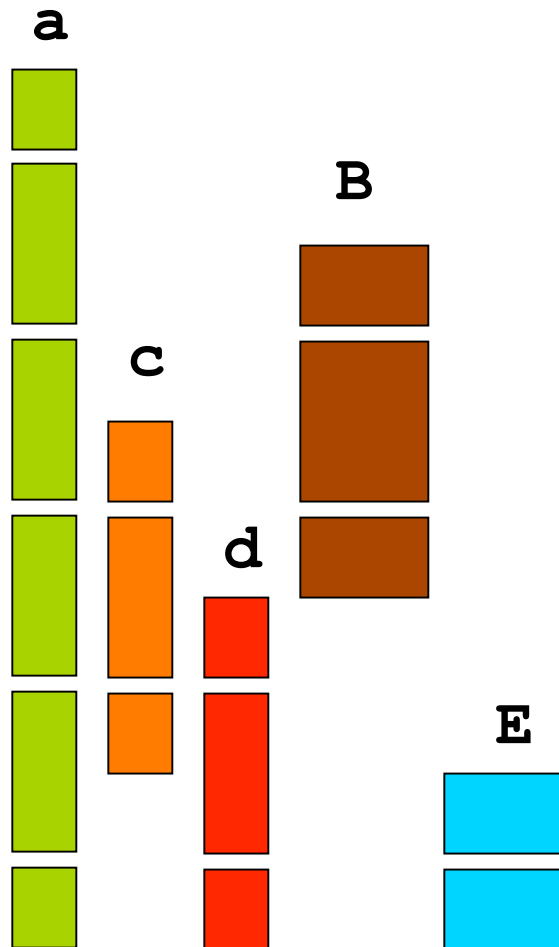


A new representation might help

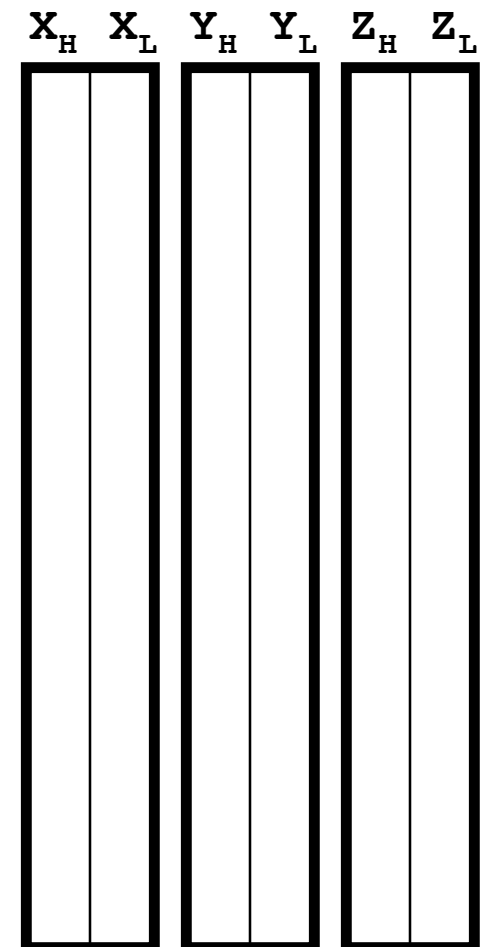
Elementary Program

$a_1 = \bullet$
 $(a_2) = (a_1)$
 $B_2 = \bullet$
 $(a_3, B_3) = (a_2, B_2)$
 $c_3 = \bullet$
 $(a_4, B_4, c_4) = (a_3, B_3, c_3)$
 $d_4 = B_4$
 $(a_5, c_5, d_5) = (a_4, c_4, d_4)$
 $E_5 = c_5$
 $(a_6, c_6, E_6) = (a_5, c_5, E_5)$
 $\bullet = a_6, d_6, E_6$

Live Ranges



Registers



And it does, indeed!

Assembly Program

$X_H = \bullet$

$Y = \bullet$

$X_L = \bullet$

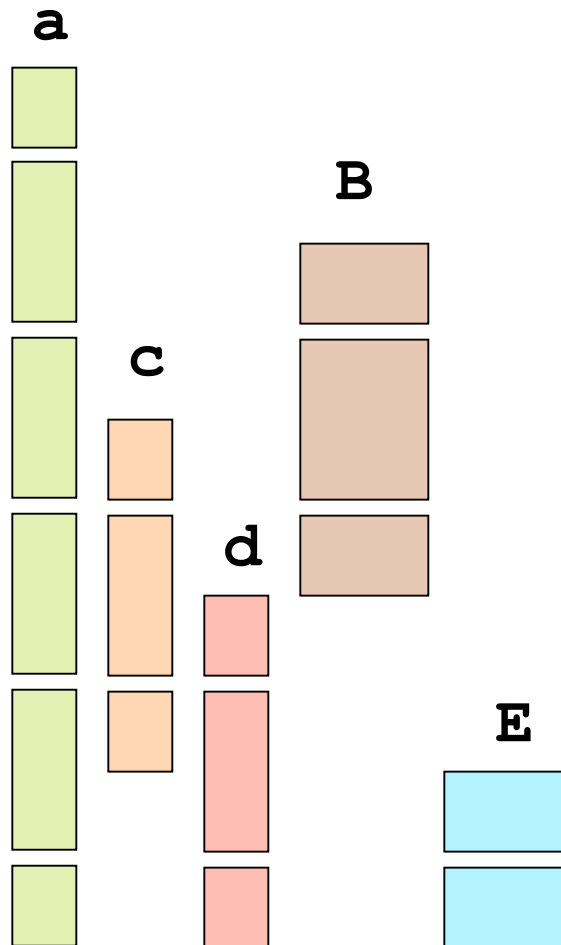
$Y_H = Y$

$Y_L = X_H$

$X = X_L$

$\bullet = Y_L, Y_H, X$

Live Ranges



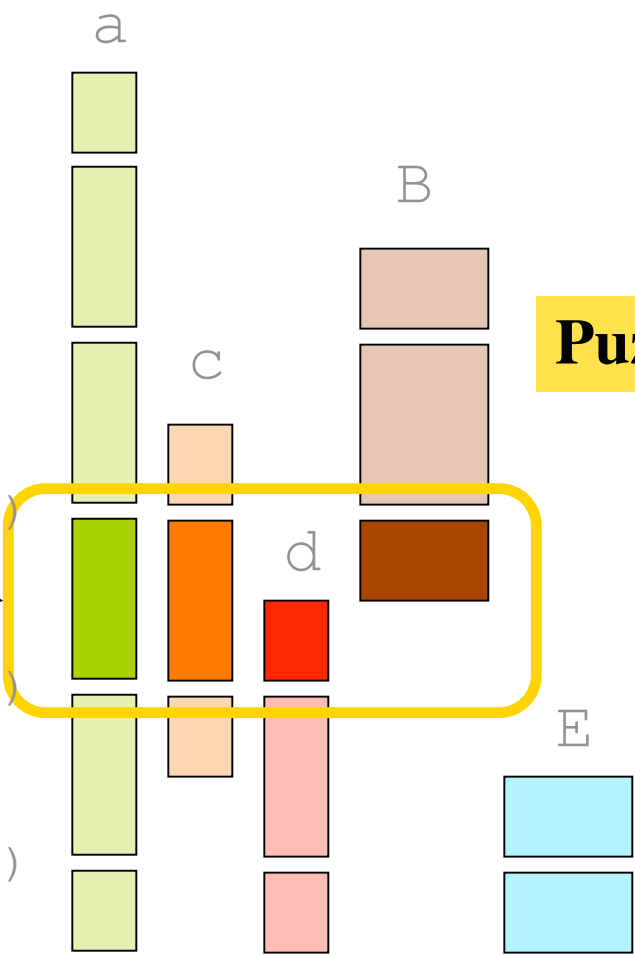
Registers



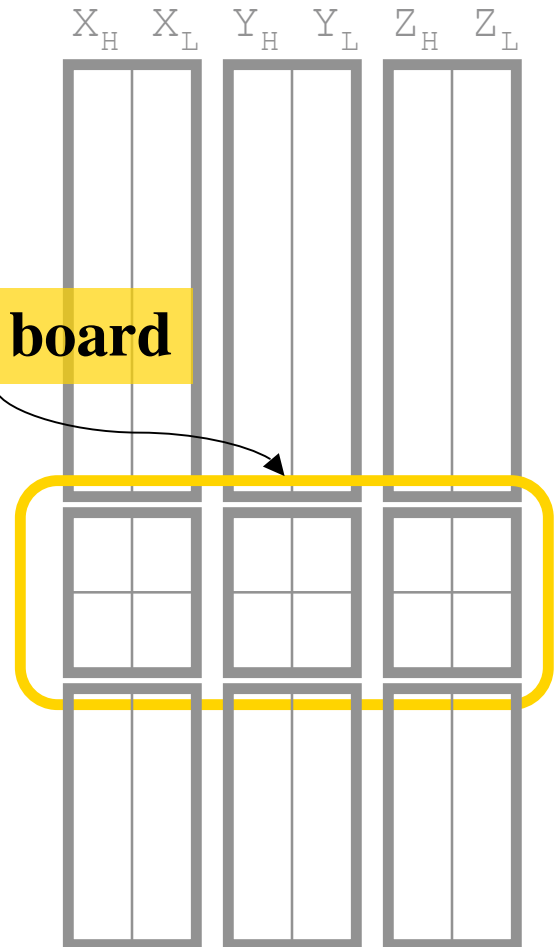
Old problem, new abstraction

$a_1 = \bullet$
 $(a_2) = (a_1)$
 $B_2 = \bullet$
 $(a_3, B_2) = (a_2, B_2)$
 $C_3 = \bullet$
 $(a_4, B_4, c_4) = (a_3, B_3, c_3)$
 $d_4 = B_4$
 $(a_5, c_5, d_5) = (a_4, c_4, d_4)$
 $E_5 = c_5$
 $(a_6, c_6, E_6) = (a_5, c_5, E_5)$
 $\bullet = a_6, d_6, E_6$

Puzzle pieces



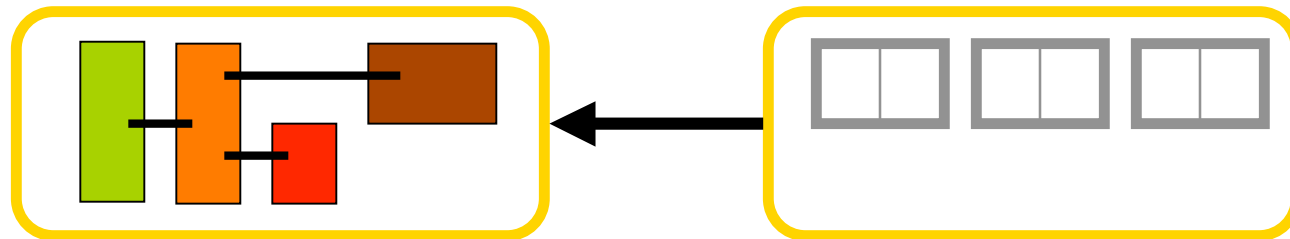
Puzzle board



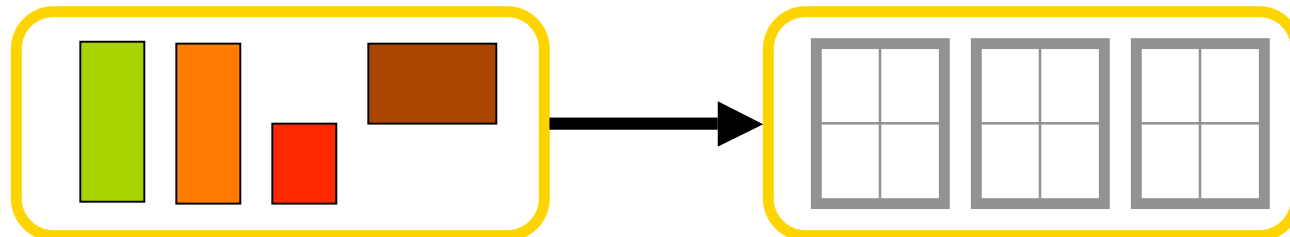
Puzzle solving is the dual of graph coloring



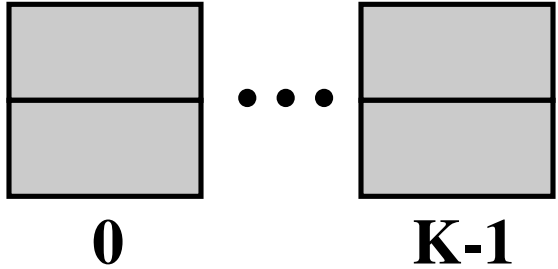
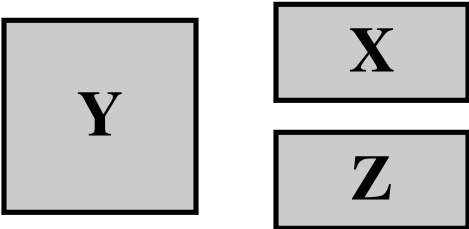
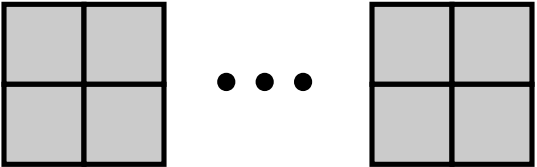
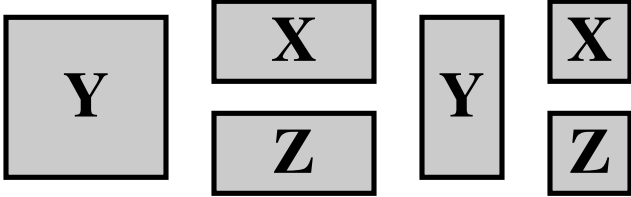
- **Graph coloring places the registers on the variables**
 - The variables form a graph
 - The registers are colors for the graph



- **Puzzle solving places the variables on the registers**
 - The registers form a board
 - The variables are pieces for the board

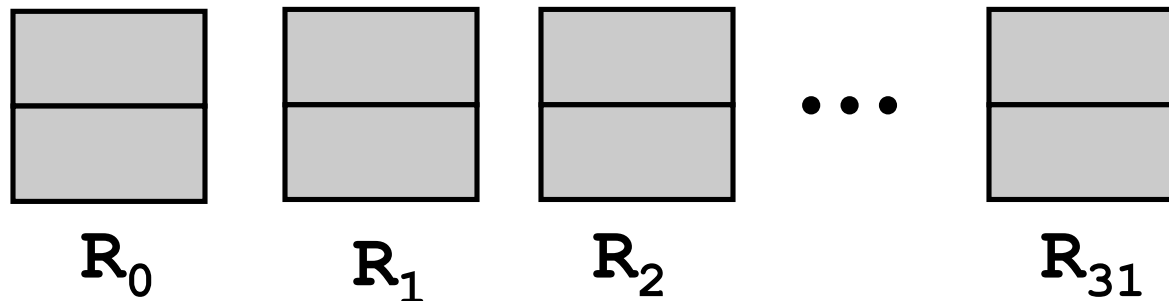


Two types of puzzles

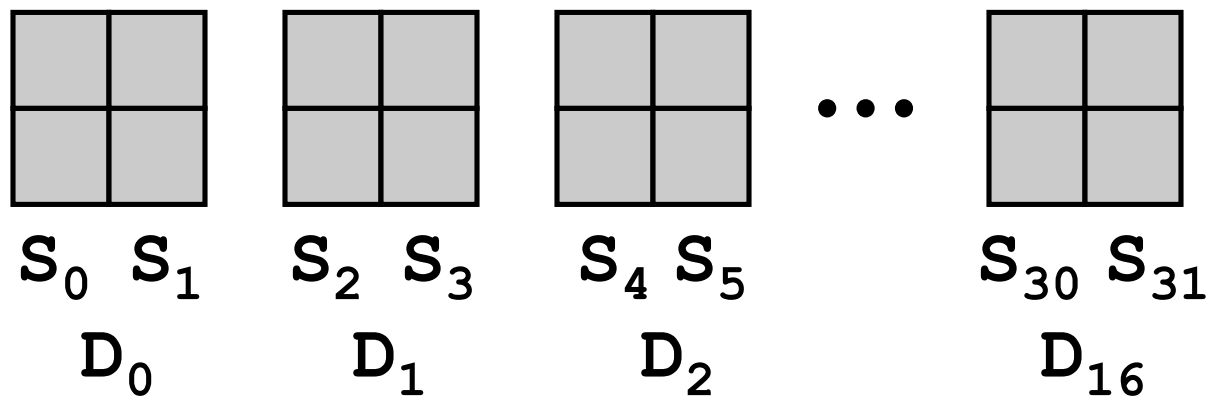
	Board	Pieces
Type-0	 <p>0 ... K-1</p>	
Type-1		

The register file determines the puzzle board

PowerPC: 32 general purpose integer registers:

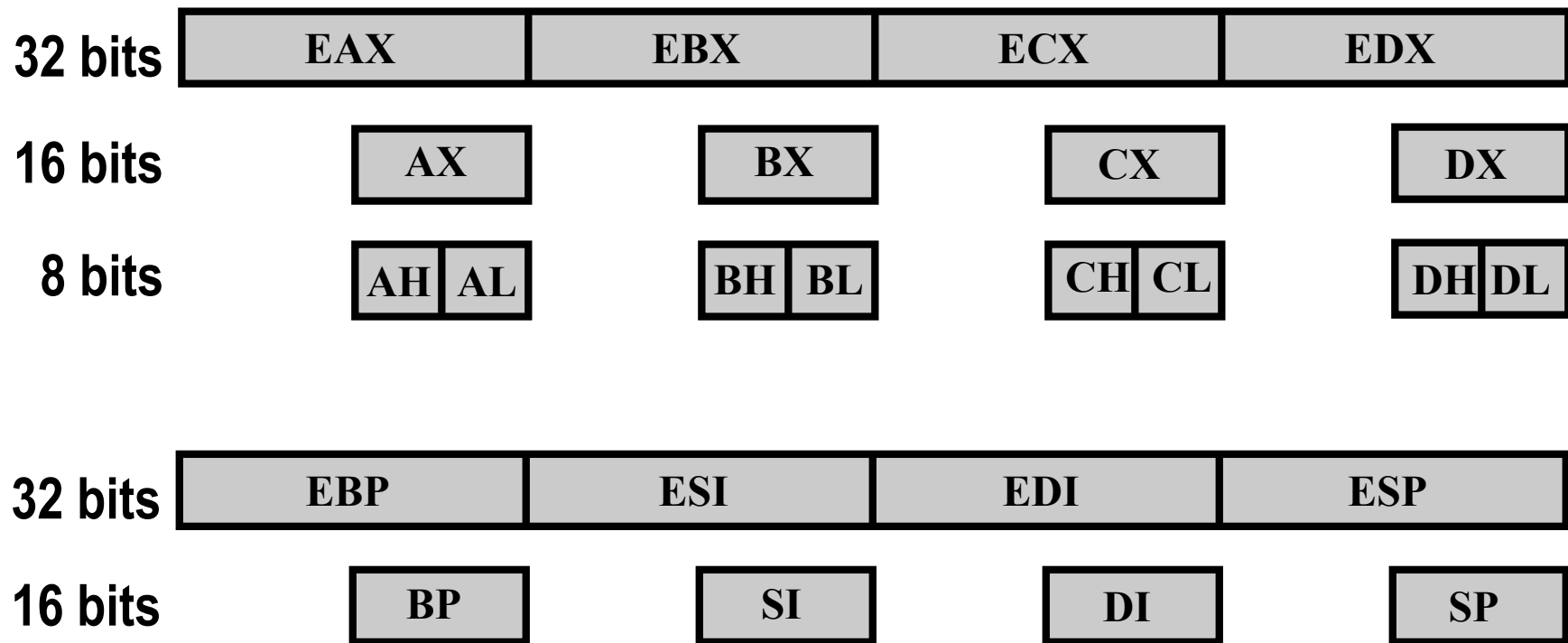


ARM: 16 double precision floating-point registers:



X86-32 register board

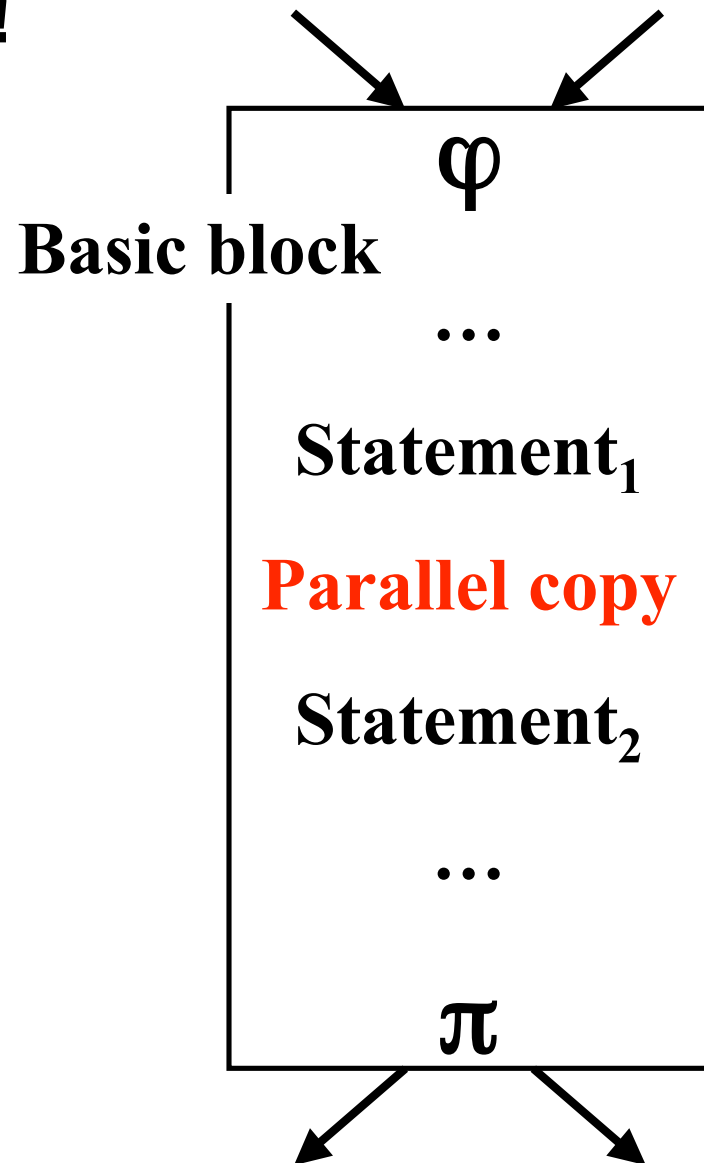
Hybrid puzzle: type-0 and type-1.



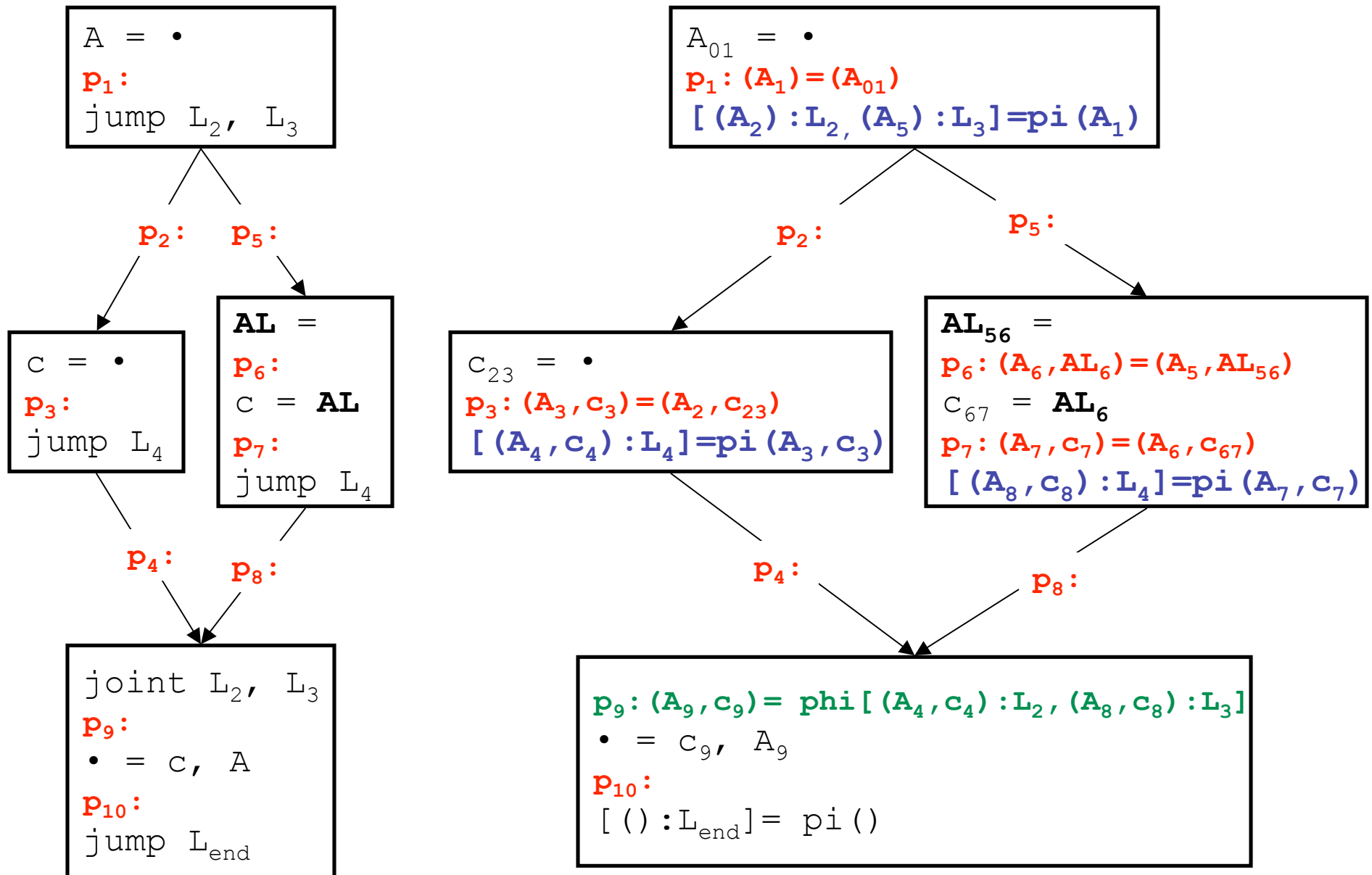
Elementary programs

Split live ranges everywhere!

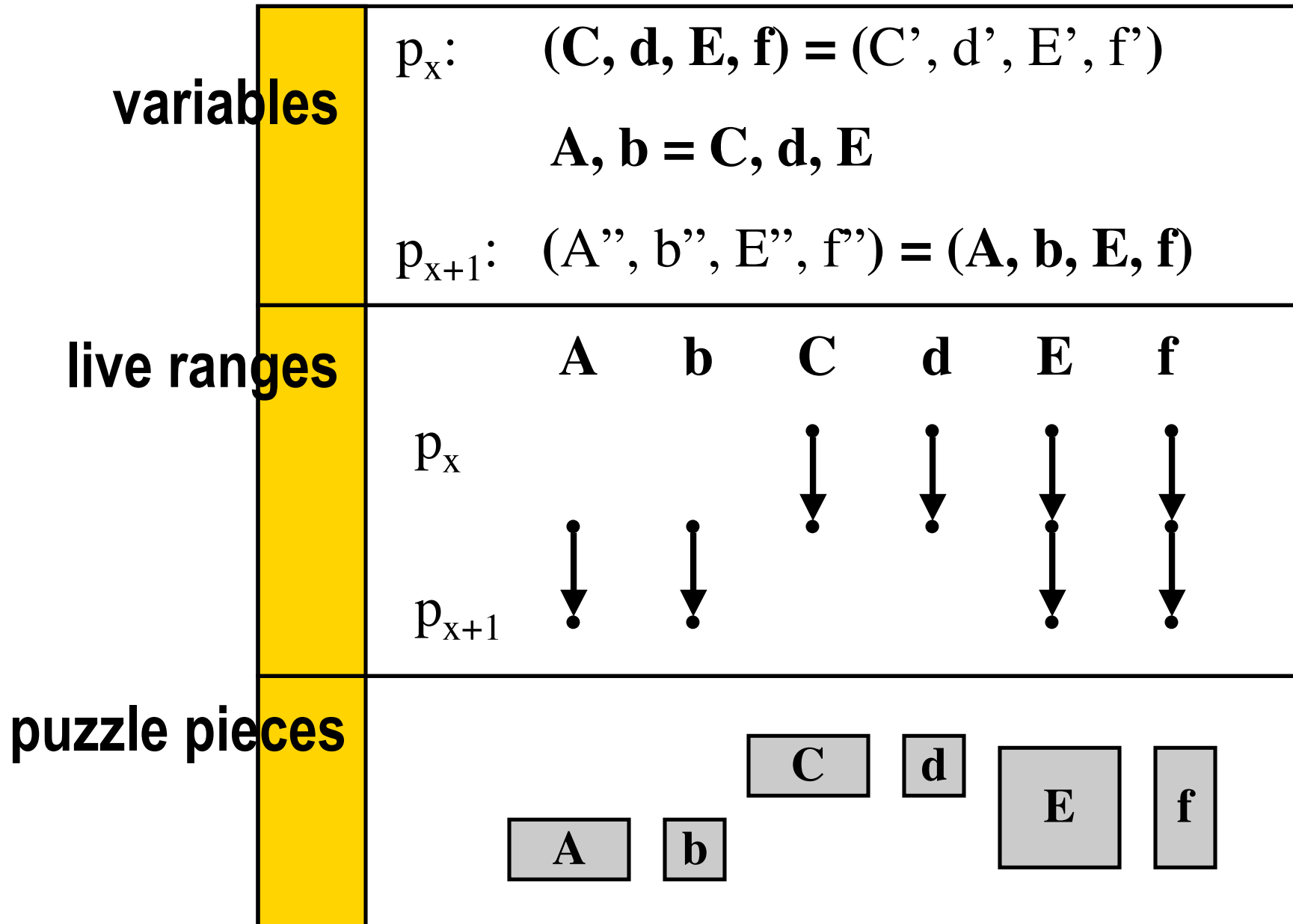
[Appel & George, PLDI 2001]



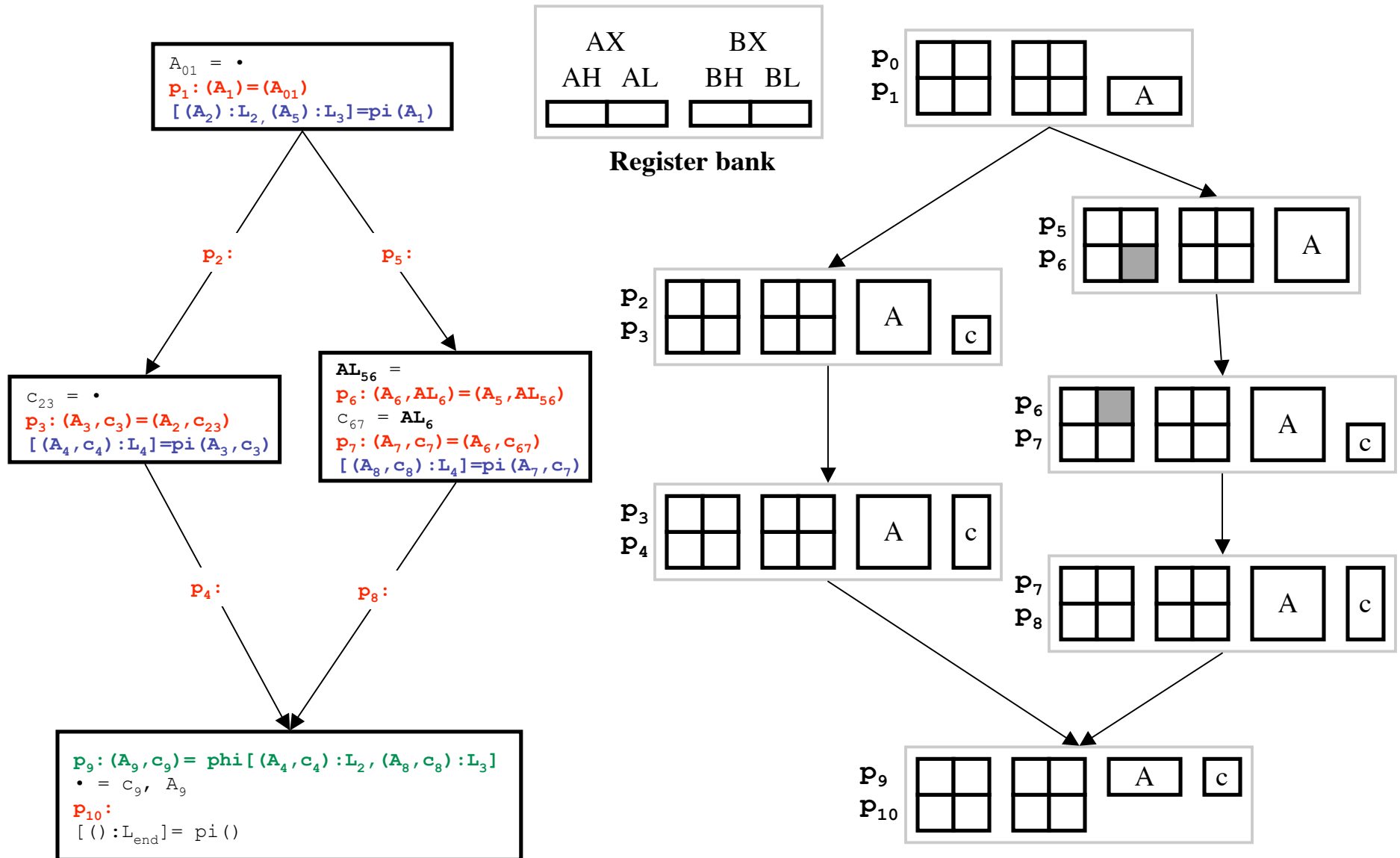
From a program to an elementary program



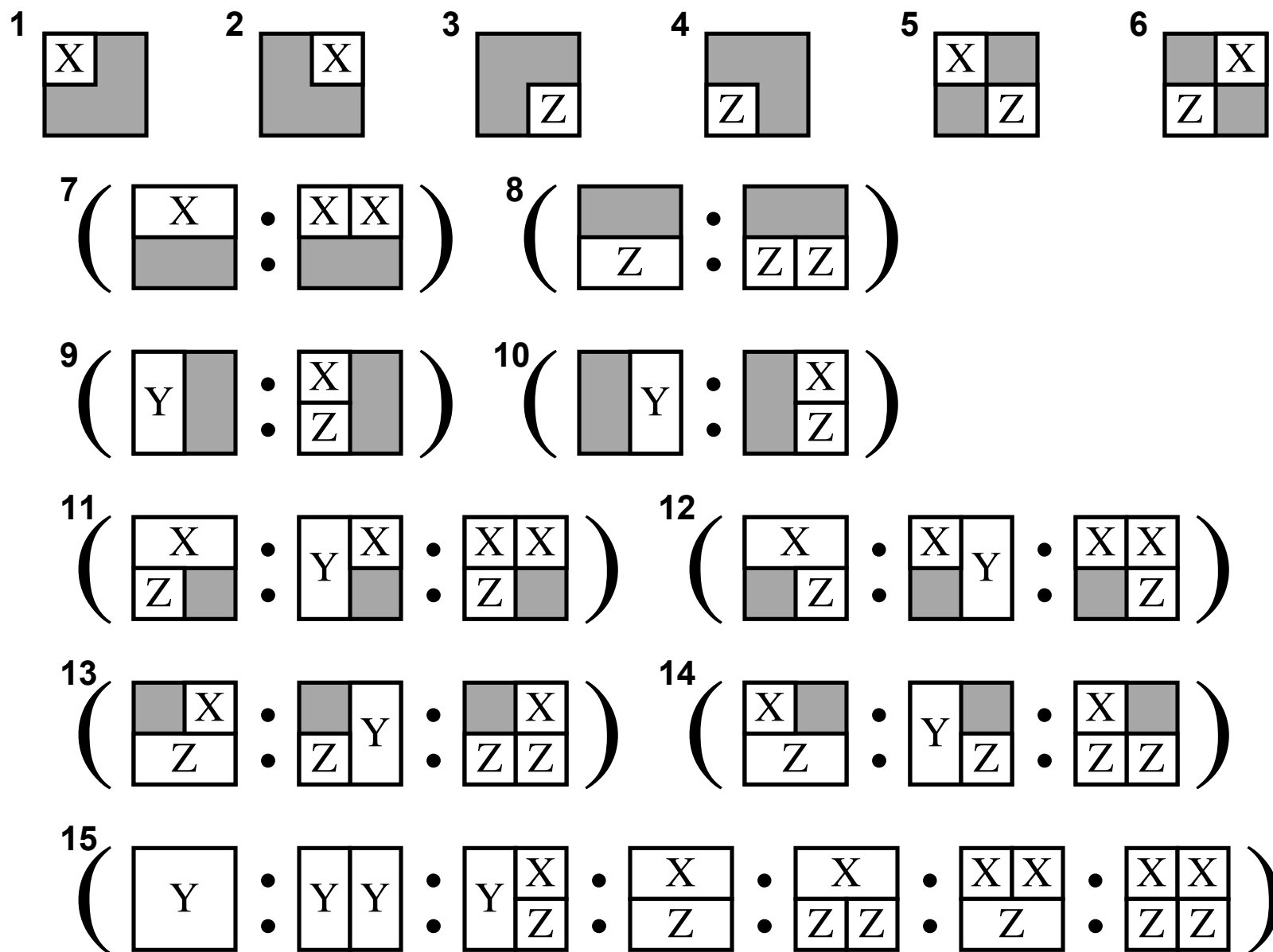
From variables to puzzle pieces



From an elementary program to puzzles

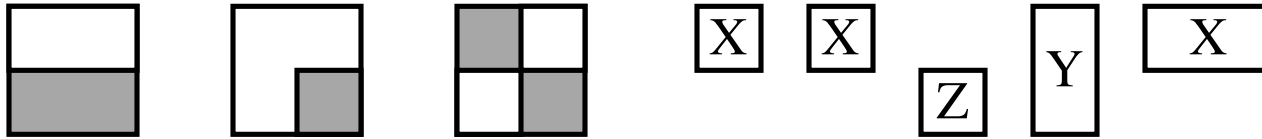


A linear algorithm for solving type-1 puzzles

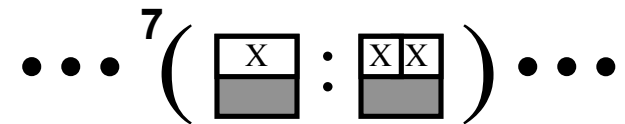


Example

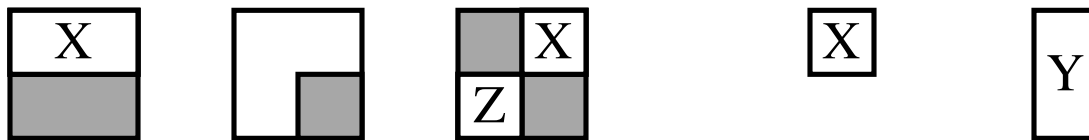
Puzzle at step 0



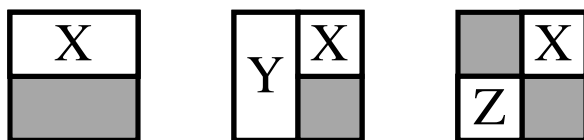
Puzzle after applying statement 6



Puzzle after applying statement 7



Puzzle after applying statement 11



Foundations

- **Theorem:** spill-free register allocation with pre-coloring for an elementary program is equivalent to solving a collection of puzzles
- **Theorem:** a puzzle is solvable if and only if our program succeeds on the puzzle
- **Theorem:** Our puzzle solving program runs in $O(\#regs)$ time

Two-phase register allocation for x86

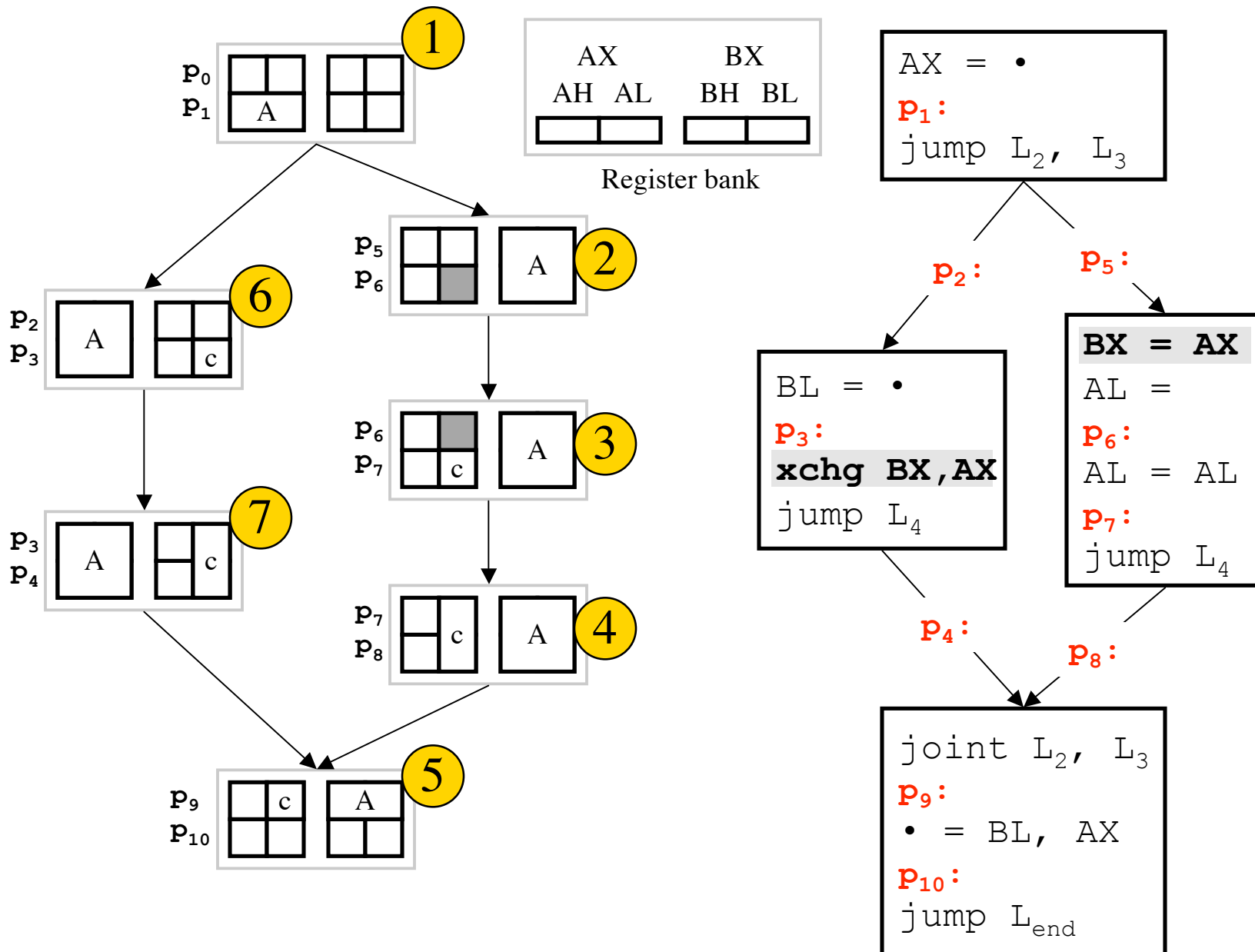
1. Spilling:

- The puzzle solver determines MaxLive
- Remove pieces until
 need-for-registers = #registers
- We use Belady's algorithm (also used in linear scan) for spilling

2. Coloring and coalescing:

- Use a puzzle solver that is guided by the solution to the previous puzzle
- If puzzle contain no pre-coloring, we can guarantee a minimal number of copies.

From puzzles to assembly code



Experimental results comparing four allocators

- **Puzzle solving**
- **Extended version of linear scan (default in LLVM)**
- **Iterated register coalescing by George & Appel, POPL 1996, with extensions by Smith, Ramsey & Holloway, PLDI 2004**
- **Partitioned boolean quadratic programs (PBQP): Scholz & Eckstein, LCTES/SCOPES 2002**
- **All four implemented in LLVM 1.9, running on an 32-bit X86**

Benchmarks: SPEC 2000, etc; total: 1.3 M LOC

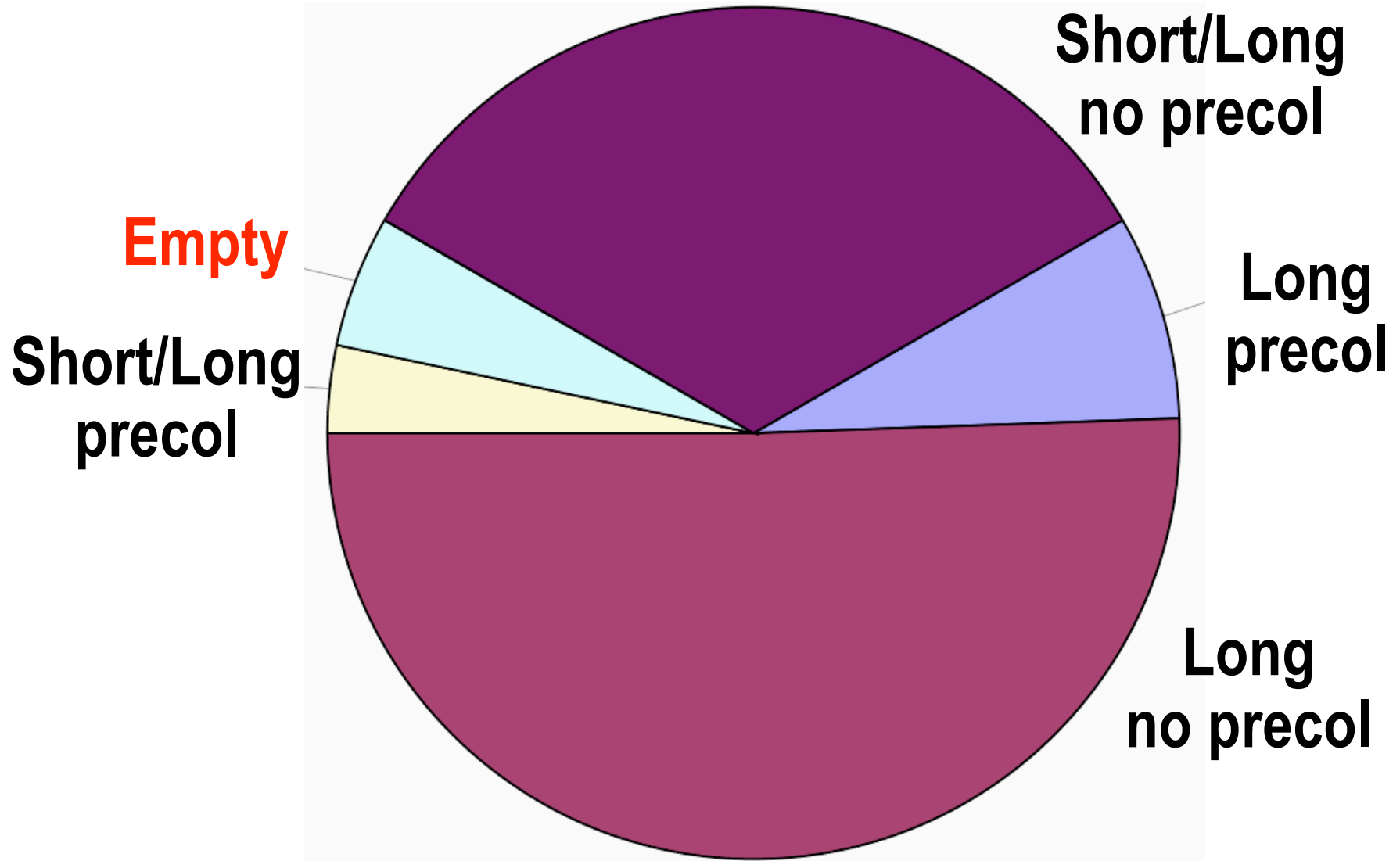
This talk:
SPEC CPU
2000

Benchmark	LoC	Asm/bytes
gcc	224,099	12,868,208
perlbmk	85,814	7,010,809
gap	71,461	4,256,317
vortex	67,262	2,714,588
mesa	59,394	3,820,633
crafty	21,197	1,573,423
(nine more)
Total	611,028	37,339,663

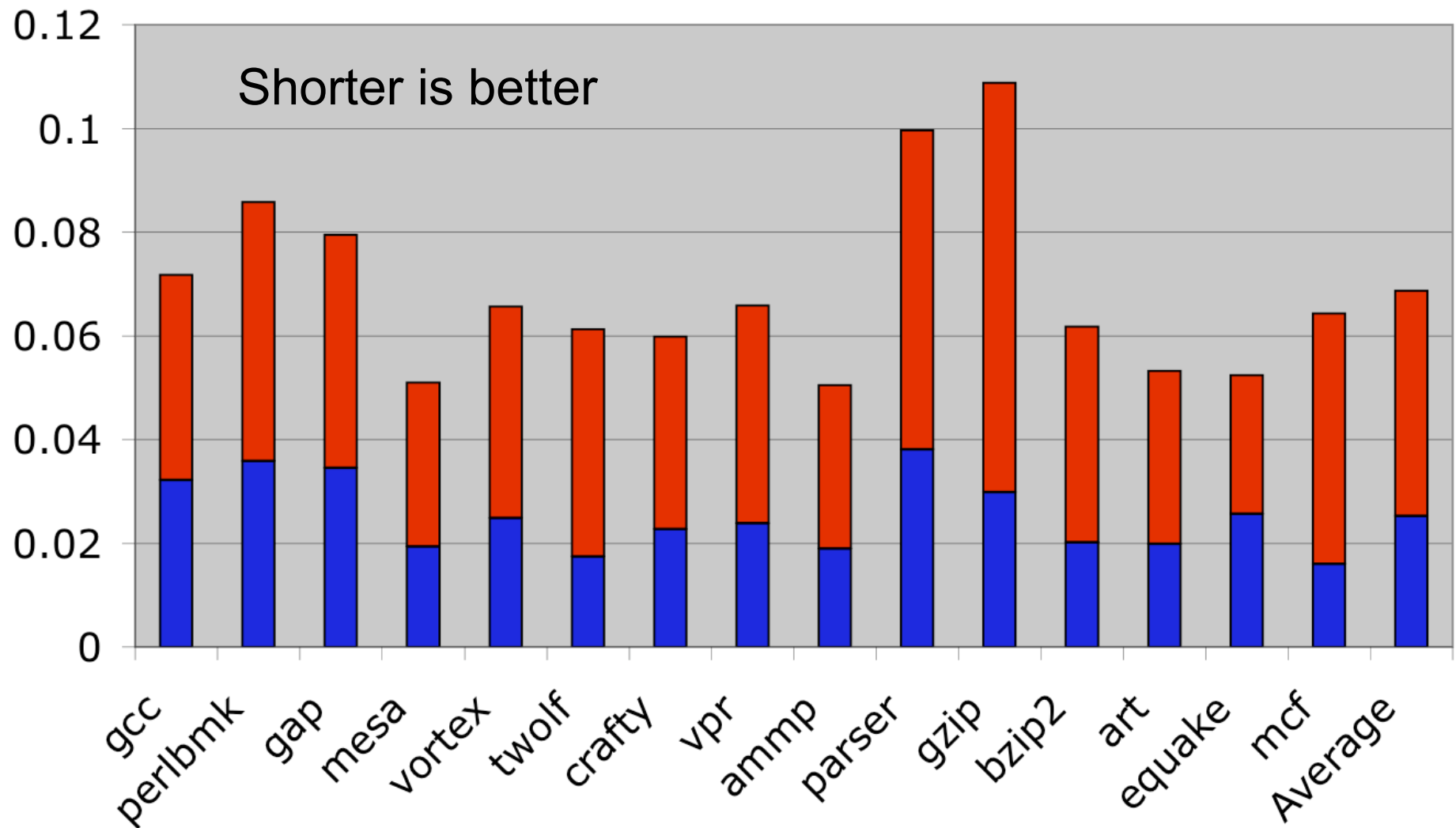
Puzzles and the number of times we solved them

Benchmark	#puzzles	avg	max	once
gcc	476,649	1.03	4	96.0%
perlbmk	265,905	1.03	4	95.4%
gap	158,757	1.05	4	96.6%
vortex	116,496	1.02	4	97.8%
mesa	139,537	1.08	9	89.7%
crafty	59,504	1.06	4	89.7%
(nine more)
Total	1,401,793	1.05	10	94.6%

A variety of puzzles

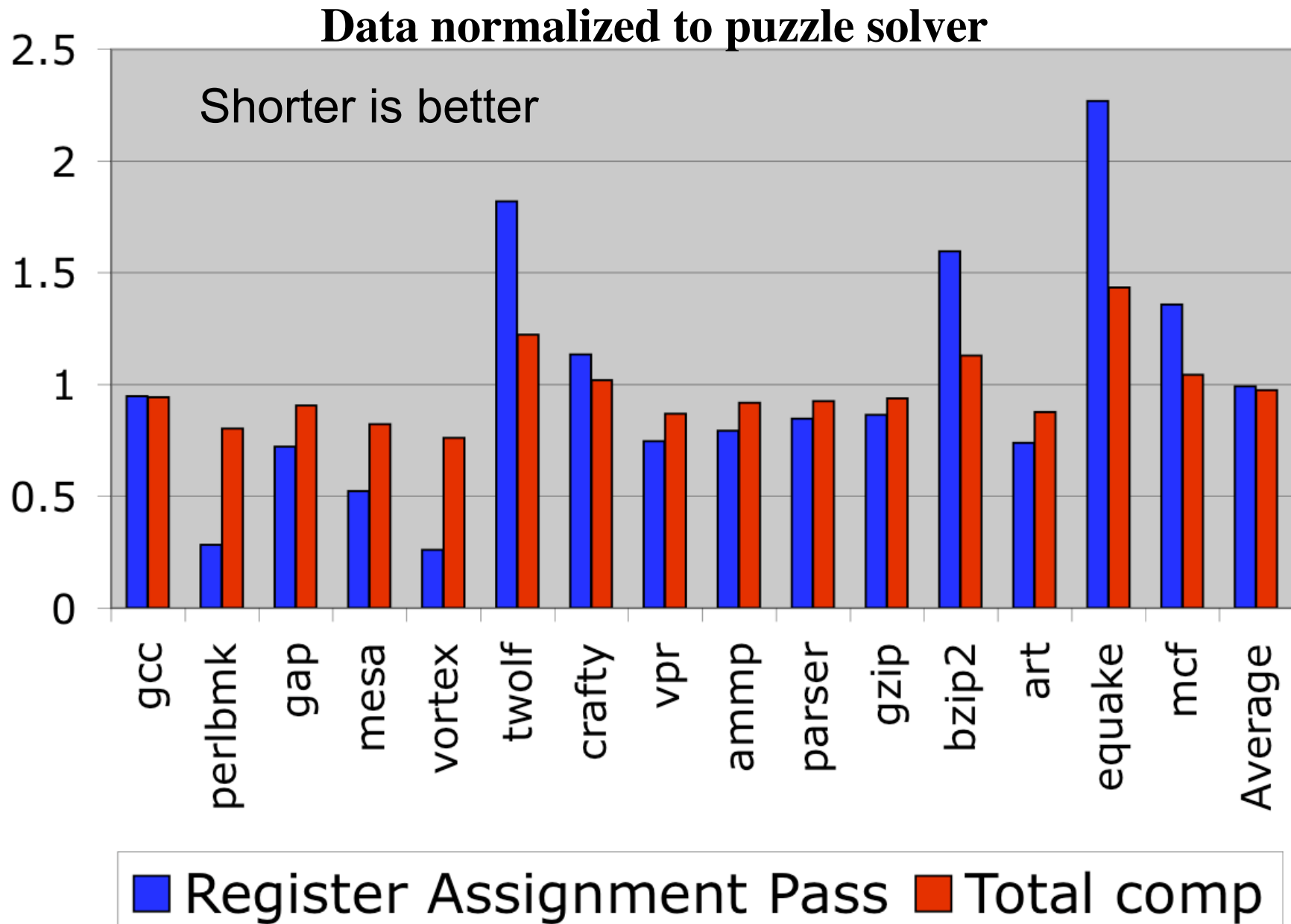


Number of moves inserted per puzzle

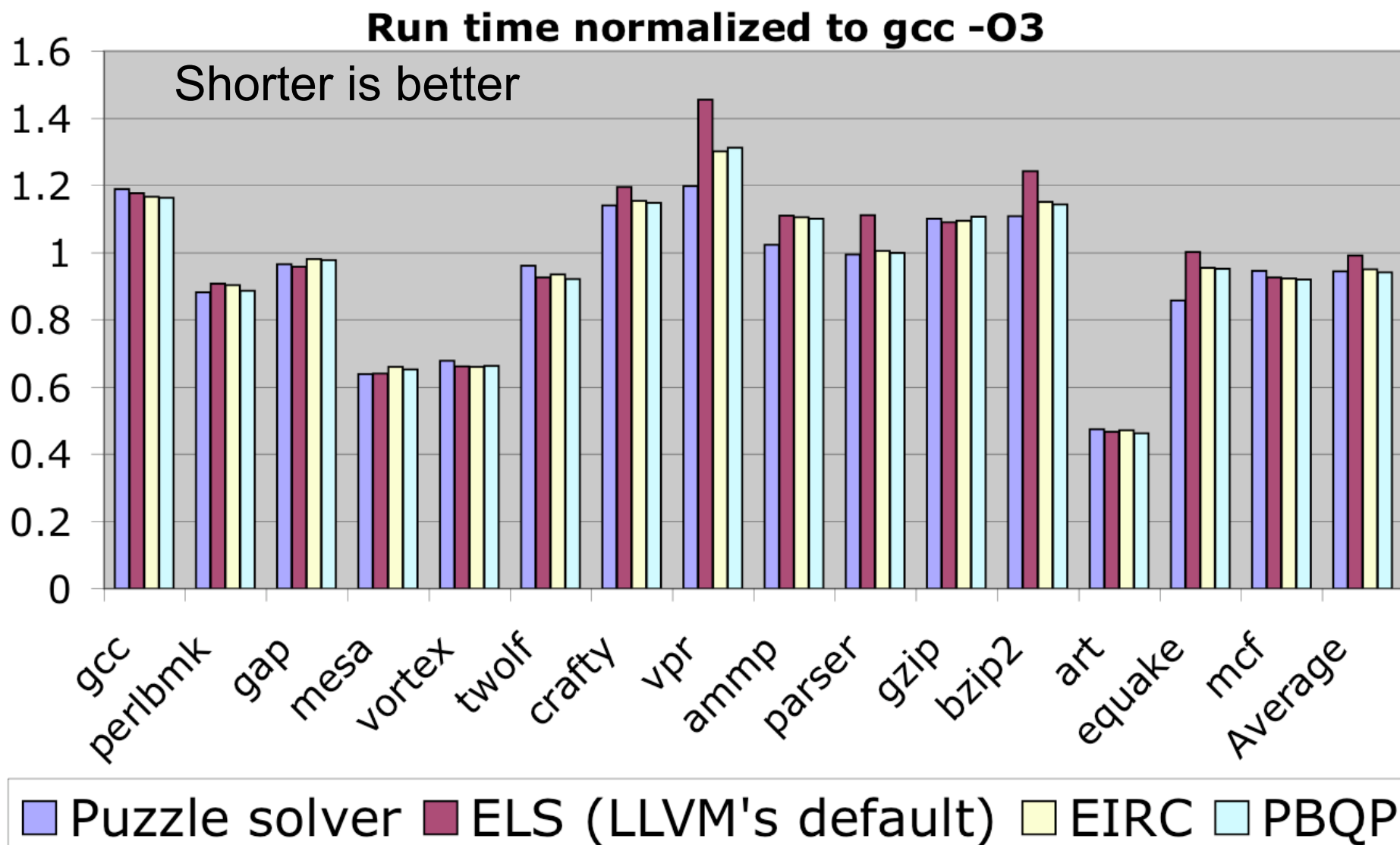


■ #(move/swaps) implementing (phi/pi)-nodes
■ #Internal Moves

Compilation time: extended linear scan vs. puzzles



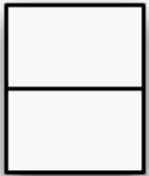
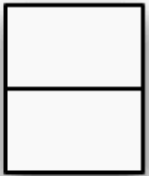


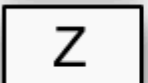
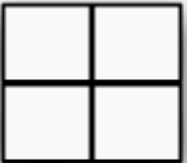
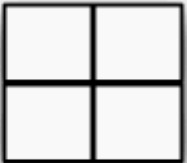

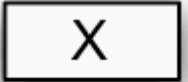


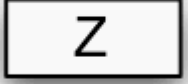












Execution time, normalized to GCC -O3



Conclusion

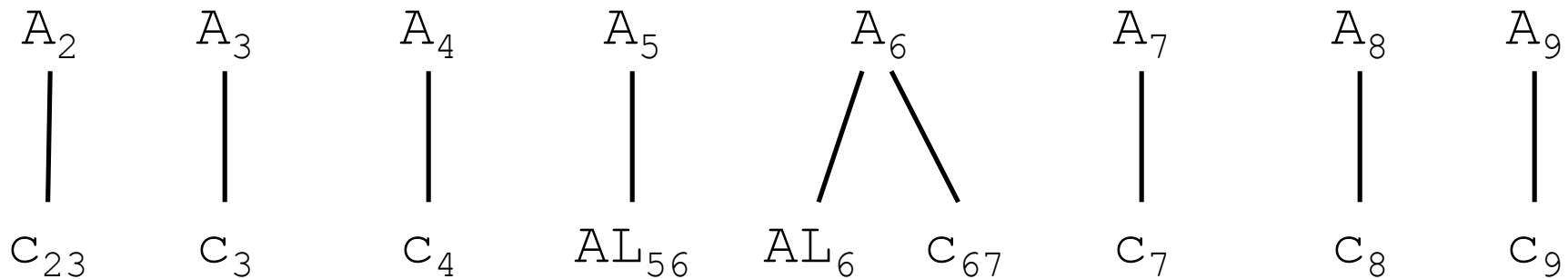
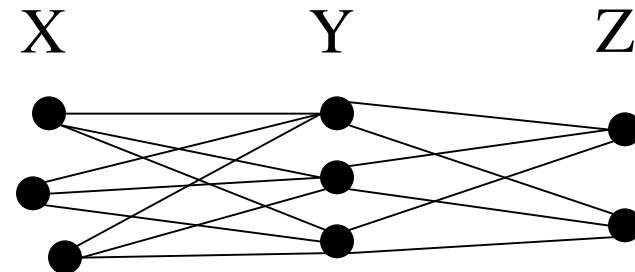
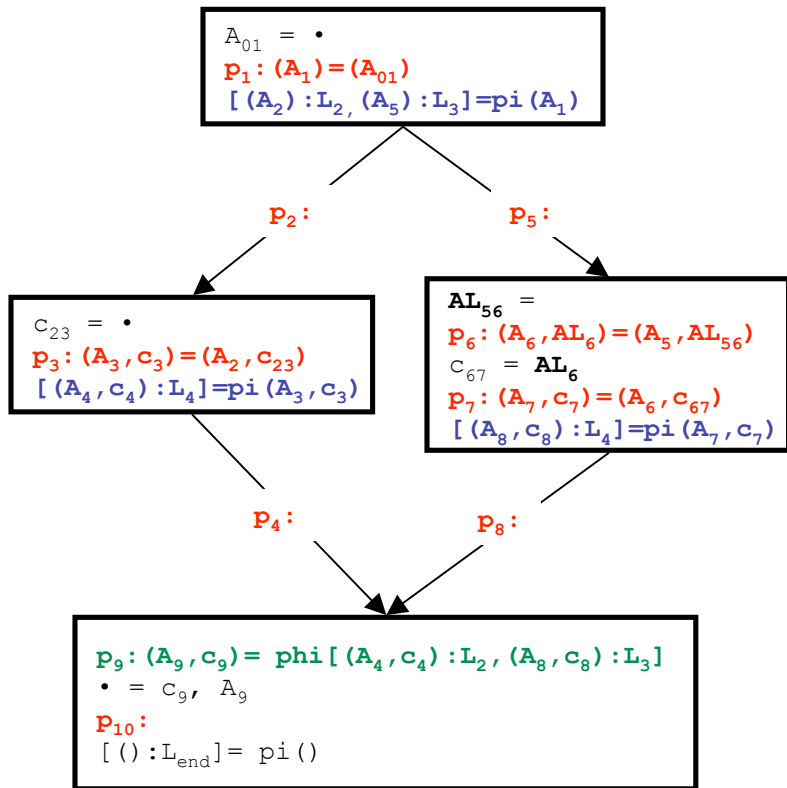
- **Register allocation by puzzle solving**
- **SSA-based register allocation for x86**
- **Handles pre-coloring, and register aliasing and pairing**
- **Split live ranges everywhere!**
- **Fast compilation time**
- **Competitive code quality**

Three types of puzzles

	Board	Kinds of Pieces
Type-0	0 $K-1$  ... 	  
Type-1	 ... 	     
Type-2	 ... 	        

Elementary programs → elementary graphs

An elementary program has an elementary interference graph



Six classes of graphs

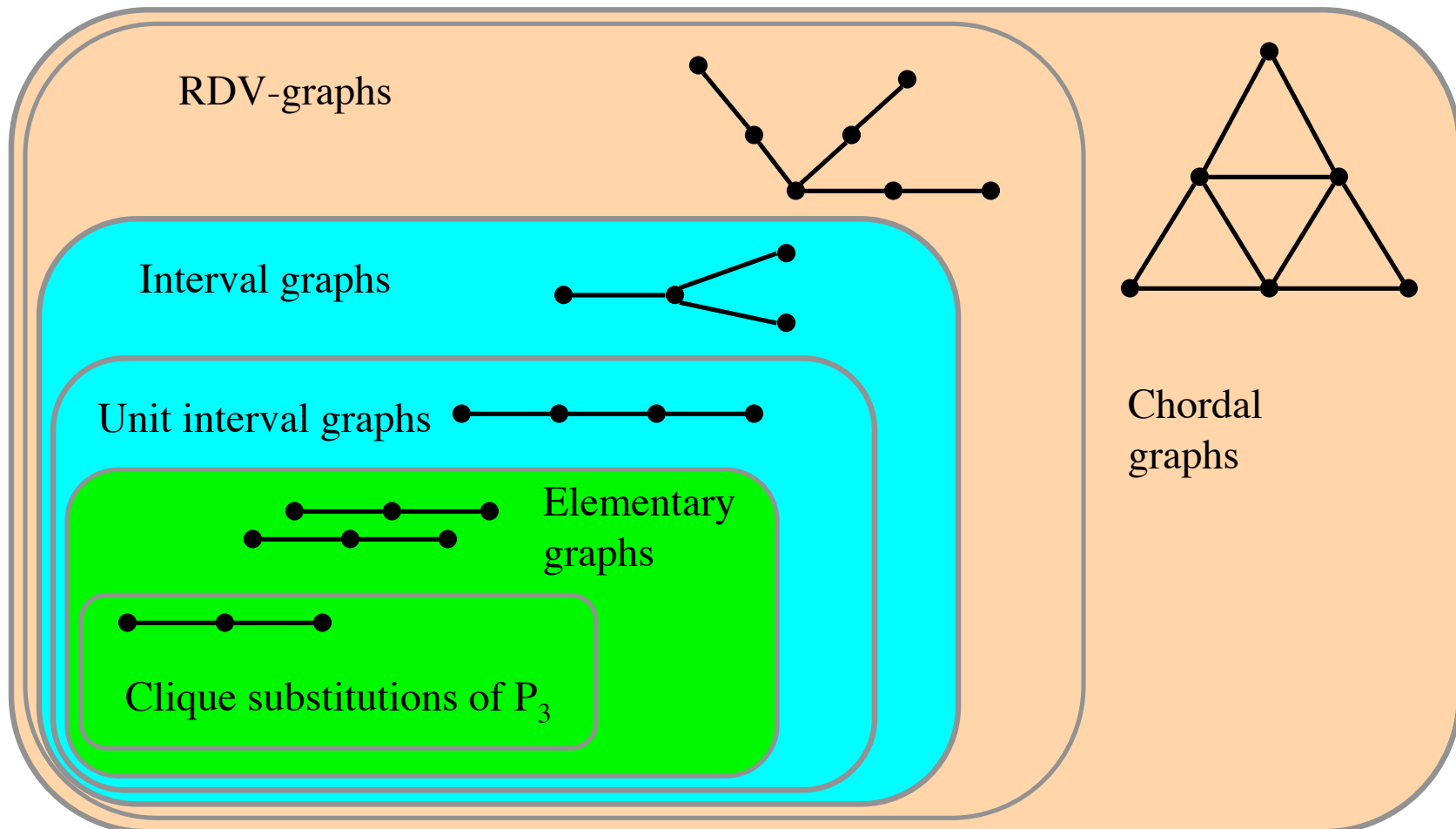
Elementary Programs

\subset

SSI-form Programs

\subset

SSA-form Programs



Complexity results for four graph problems

Problem	General	Chordal	Interval	Elementary
Aligned 1-2-coloring extension	NP-complete	NP-complete	NP-complete	Linear time [this paper]
Aligned 1-2-coloring	NP-complete	NP-complete	NP-complete	Linear time [this paper]
Coloring extension	NP-complete	NP-complete	NP-complete	Linear time [this paper]
Coloring	NP-complete	Linear time	Linear time	Linear time