Update:
PBQP Register Allocation

PBQP Background

- Born out of DSP compiler research
- Easily support complex constraints
- Implemented in LLVM in version 2.4

Cost Model

For each variable, a 1D table describes costs:

SPILL	50
R0	0
R1	5

Cost Model

For pairs of variables… 2D tables describes costs:

SPILL	50
R0	0
R1	5

	SPILL	R0	R1
SPILL	0	0	0
R0	0	∞	0
R1	0	0	∞

50	SPILL
0	R0
5	R1

Built-in Constraints

- Spill costs
- Interference
- Coalescing

You can add your own costs on top...

Use case 1

- An extremely small CPU:
  - 16-bits instructions set
  - 16 x 16-bits integer registers
- Many 4-register-operands domain specific instructions:

```
instr reg1, reg2, reg3, reg4
```

- Can not be encoded in 16 bits, so need some *pairing*:

```
instr reg(I), reg(J)[, reg(I+1), reg(J+1)]
```

Use case 1: coalescing

- Pairing requires *different* registers
- But the coalescer will have happily coalesced registers in a pair if they hold the same value
- So the first step was to *smartly* undo the coalescer's work:
  - insert register copies --- only where really needed
  - do not forget to update liveness info !

Use case 1: constraints

- Obvious pairing constraint:

  reg(I+1) is the successor of reg(I)

- The pairing constraint is transitive !

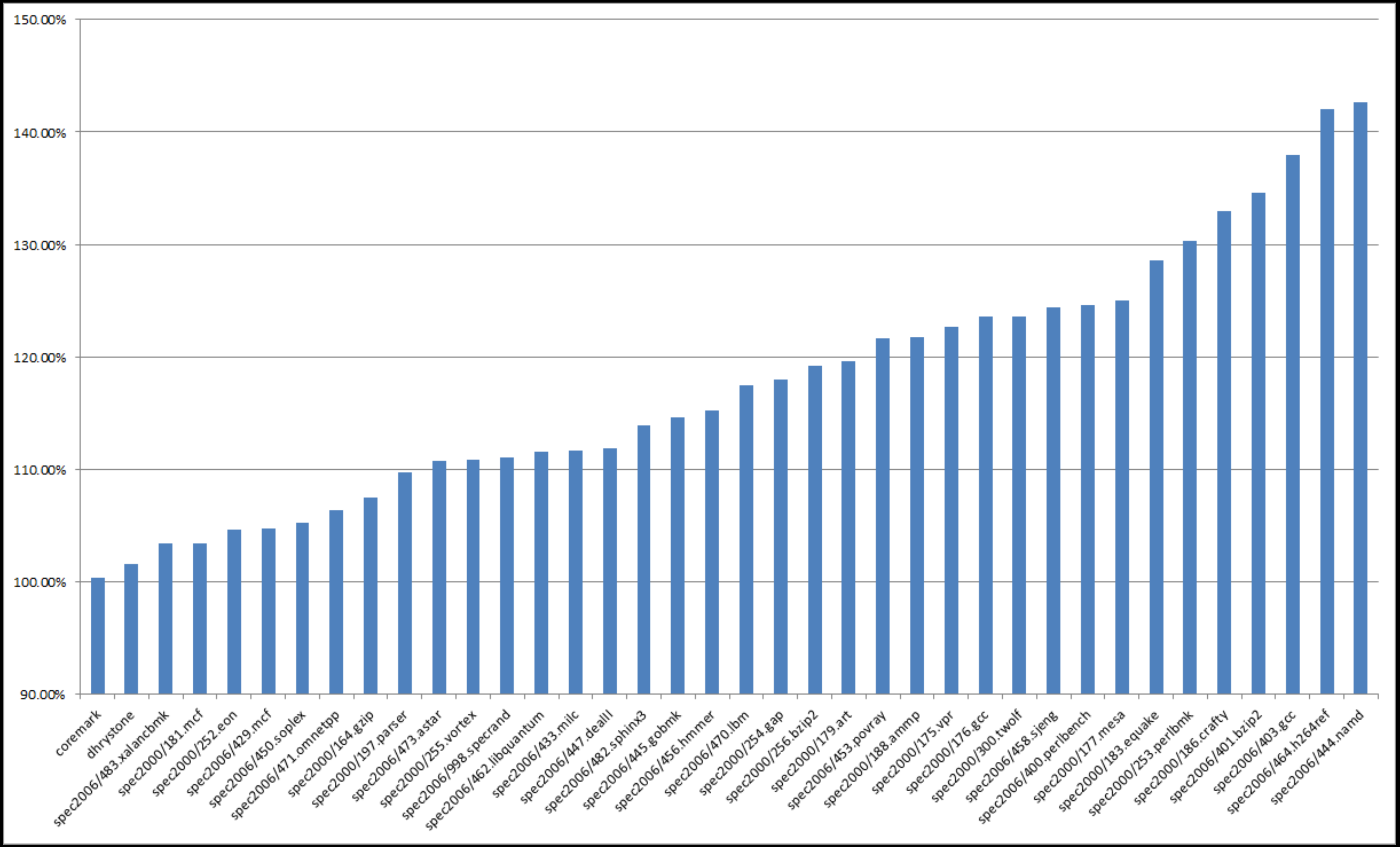
  reg(I) == reg(J+1) => pair2(reg(I+1), reg(J))

- The constraint set must be complete ... or things will break

Recent Work

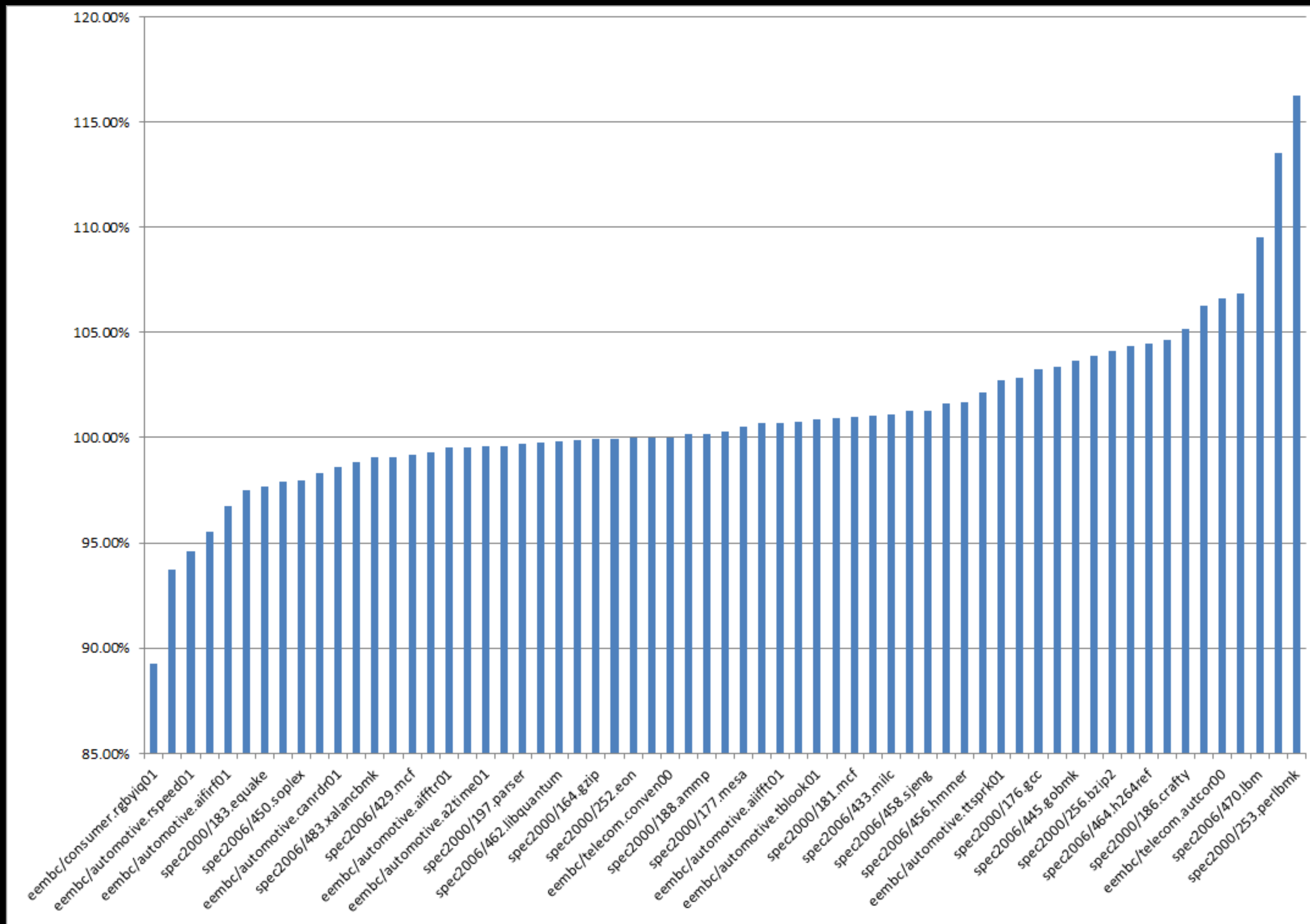
- Easier to use with composable constraints
- Solver improvements
  - Reductions in memory consumption
  - Reductions in compile time
  - Improvements in allocation quality

AArch64: compilation time PBQP / Greedy



PBQP is 17% slower than Greedy

AArch64: execution time PBQP / Greedy



PBQP is 1% slower than Greedy

Try It Out

- Competitive code quality / performance
- Easy to customize
- Do not hesitate to talk to us:
  - Arnaud: arnaud.degrandmaison@arm.com
  - Lang: lhames@gmail.com

Iterative Register Allocation