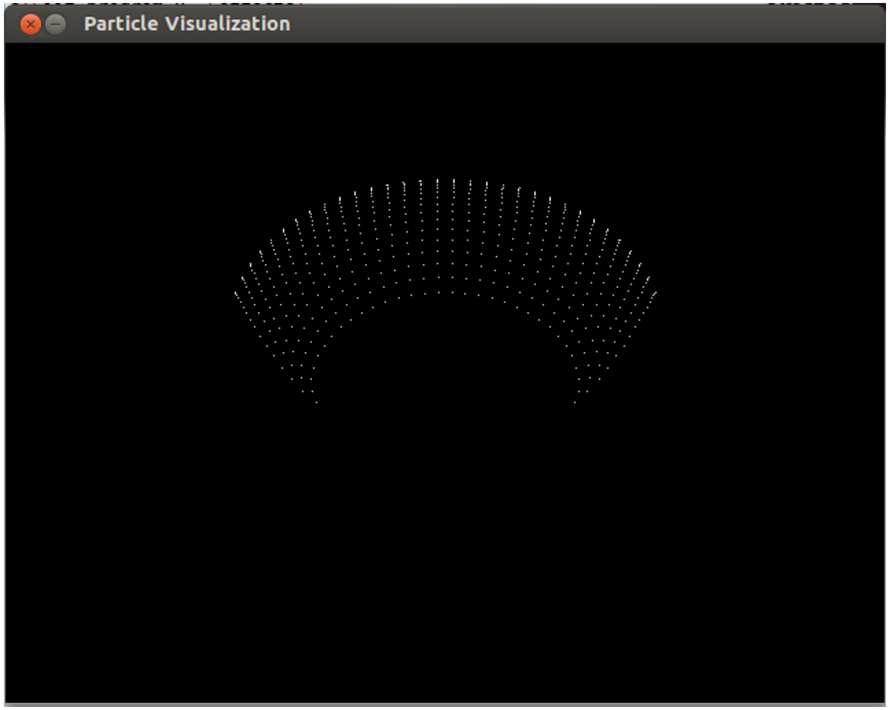
Dynamically Controlled Particle Systems

Jakob Progsch
Simon Laube
Benjamin Flück

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Particle Systems

- **Simulate a number of particles according to some given rules**
- **Particles are represented by**
  - Position[3]
  - Mass
  - Velocity[3]
  - Charge
- **Rules define forces and effects**
  - Gravity
  - Collisions
  - Time step in the simulation
  - …



Dynamic Control

- **Algorithm:**

  Iterate over all particles and apply the desired rules

- **Problem:**

  Changing rules in a dynamic/random environment does not allow for many optimizations during compilation (e.g. games) between the selected rules

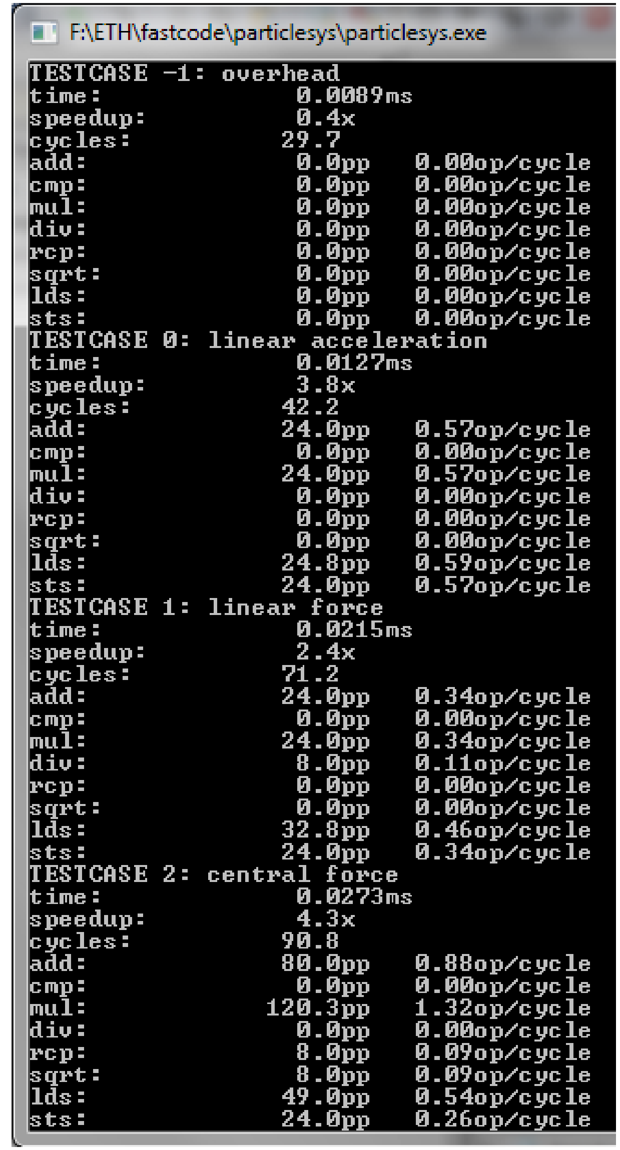
- **Our Solution:**

  Create a JIT-compiler to optimize these rules on the fly when they change

Performance Measurements

- **Performance in flops/particle**
  - Number of particles has no influence;
    linear iteration through particles
    -> prefetcher hides all ram accesses

- **No flops/cycle (available, but unused)**
  - Not useful for evaluating break-even point for JIT compiler
  - JIT compiler changes number of ops, stores, and loads
  - Branching/masking further skews results



```
F:\ETH\fastcode\particlesys\particlesys.exe
TESTCASE -1: overhead
time:              0.0089ms
speedup:           0.4x
cycles:           29.7
add:               0.0pp    0.00op/cycle
cmp:               0.0pp    0.00op/cycle
mul:               0.0pp    0.00op/cycle
div:               0.0pp    0.00op/cycle
rcp:               0.0pp    0.00op/cycle
sqrt:              0.0pp    0.00op/cycle
lds:               0.0pp    0.00op/cycle
sts:               0.0pp    0.00op/cycle
TESTCASE 0: linear acceleration
time:              0.0127ms
speedup:           3.8x
cycles:           42.2
add:              24.0pp    0.57op/cycle
cmp:               0.0pp    0.00op/cycle
mul:              24.0pp    0.57op/cycle
div:               0.0pp    0.00op/cycle
rcp:               0.0pp    0.00op/cycle
sqrt:              0.0pp    0.00op/cycle
lds:              24.8pp    0.59op/cycle
sts:              24.0pp    0.57op/cycle
TESTCASE 1: linear force
time:              0.0215ms
speedup:           2.4x
cycles:           71.2
add:              24.0pp    0.34op/cycle
cmp:               0.0pp    0.00op/cycle
mul:              24.0pp    0.34op/cycle
div:               8.0pp    0.11op/cycle
rcp:               0.0pp    0.00op/cycle
sqrt:              0.0pp    0.00op/cycle
lds:              32.8pp    0.46op/cycle
sts:              24.0pp    0.34op/cycle
TESTCASE 2: central force
time:              0.0273ms
speedup:           4.3x
cycles:           90.8
add:              80.0pp    0.88op/cycle
cmp:               0.0pp    0.00op/cycle
mul:             120.3pp    1.32op/cycle
div:               0.0pp    0.00op/cycle
rcp:               8.0pp    0.09op/cycle
sqrt:              8.0pp    0.09op/cycle
lds:              49.0pp    0.54op/cycle
sts:              24.0pp    0.26op/cycle
```

Baseline Implementation

- List of function pointers, sequentially applied to all particles
- Example rule: gravitation

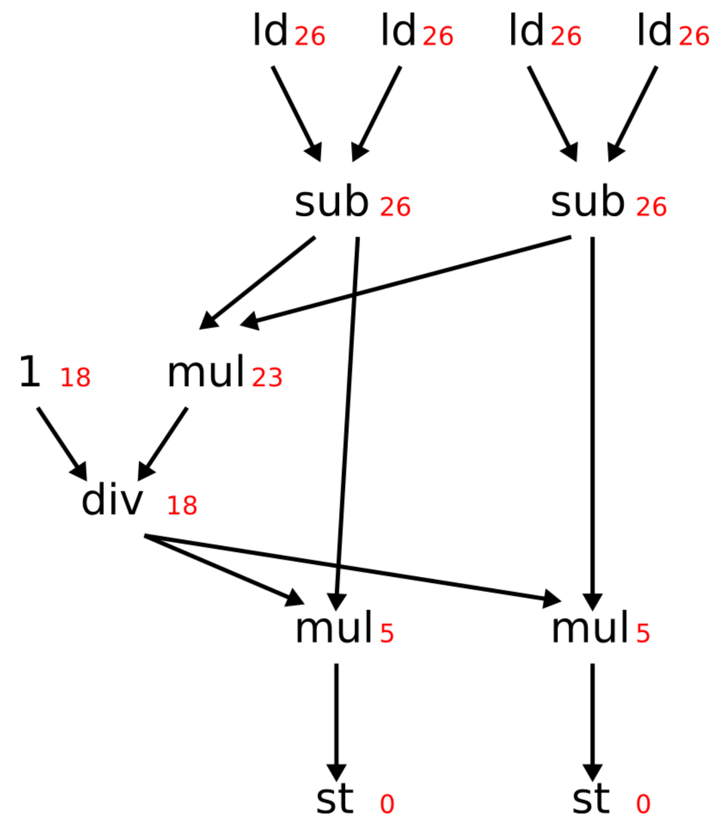
```
gravitational_force_apply(particle p, centre c, float dt) {
       diff = p.position – c
       r = distance(p.position, c)
       r3 = 1.0f/(r*r*r)
       p.velocity += dt*mu*diff*r3
}
```

Optimizations

- **Standard optimizations from class**
  - Scalar replacement
  - Loop unrolling
  - Compiler optimizations
- **SIMD optimizations on a per-rule basis**
  - SSE intrinsics
- **JIT compilation for global optimizations across rules including:**
  - SSE and AVX Code generator
  - Load/Store combining
  - Constant folding
  - Common sub-expression elimination
  - Scheduling

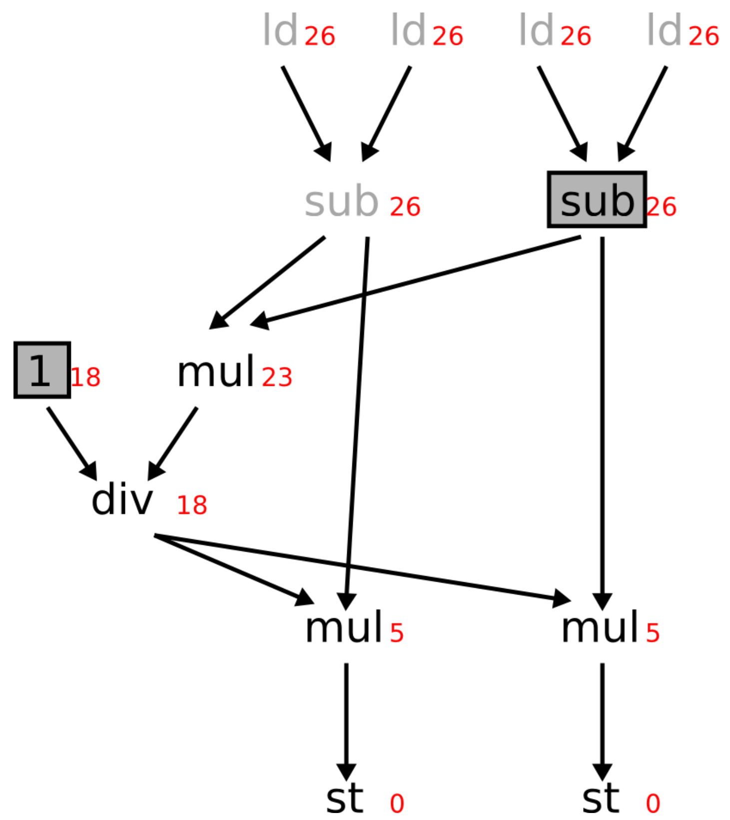
Optimizations – JIT Scheduling

- Build a dependency DAG
- Sum up latencies



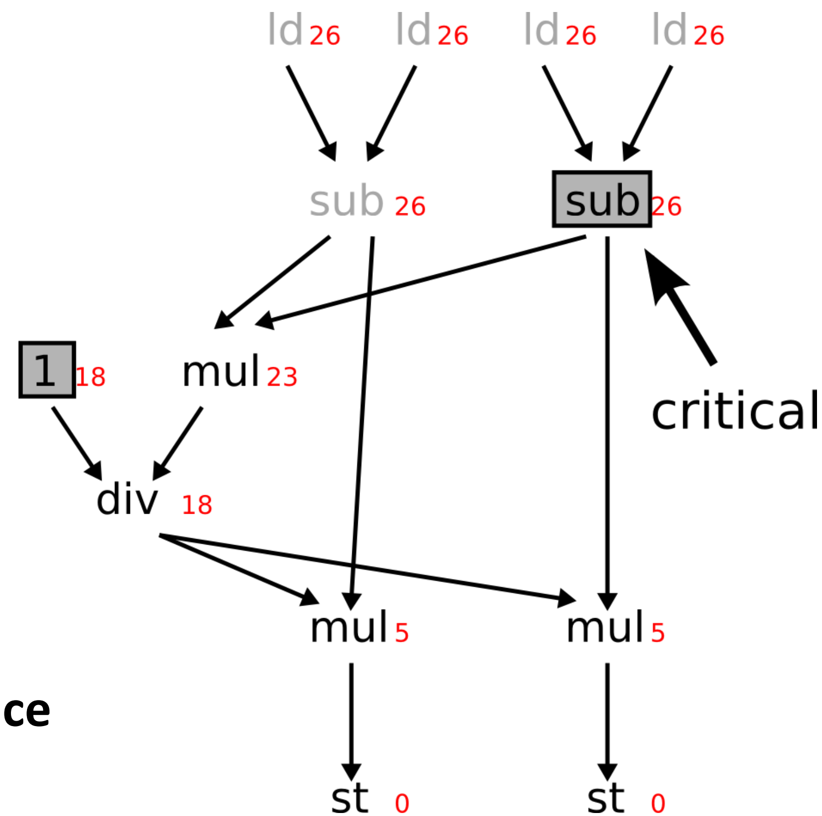
Optimizations – JIT Scheduling

- **Build a dependency DAG**
- **Sum up latencies**
- **Schedule instruction with highest latency**



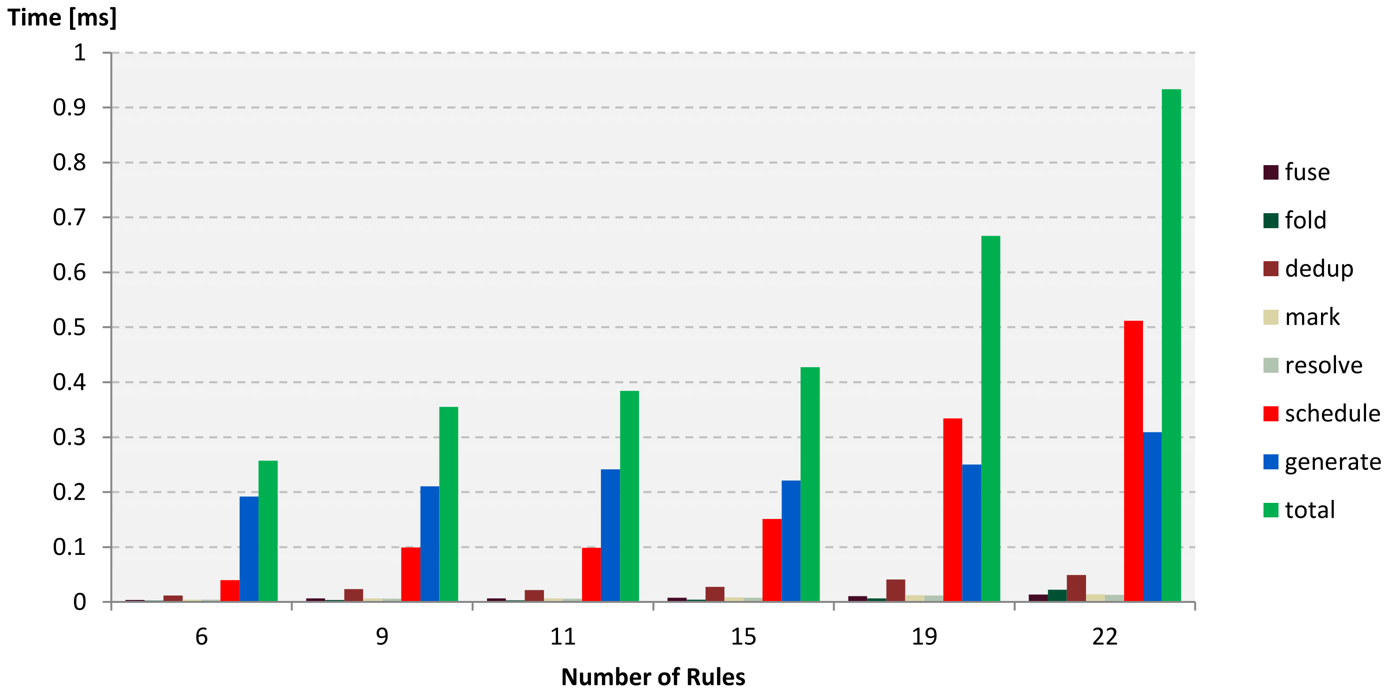
Optimizations – JIT Scheduling

- Build a dependency DAG
- Sum up latencies
- Schedule instruction with highest latency
- -> Ops on critical path get preferential treatment
- Worst case runtime is O(op^2)
  In practice close to linear performance



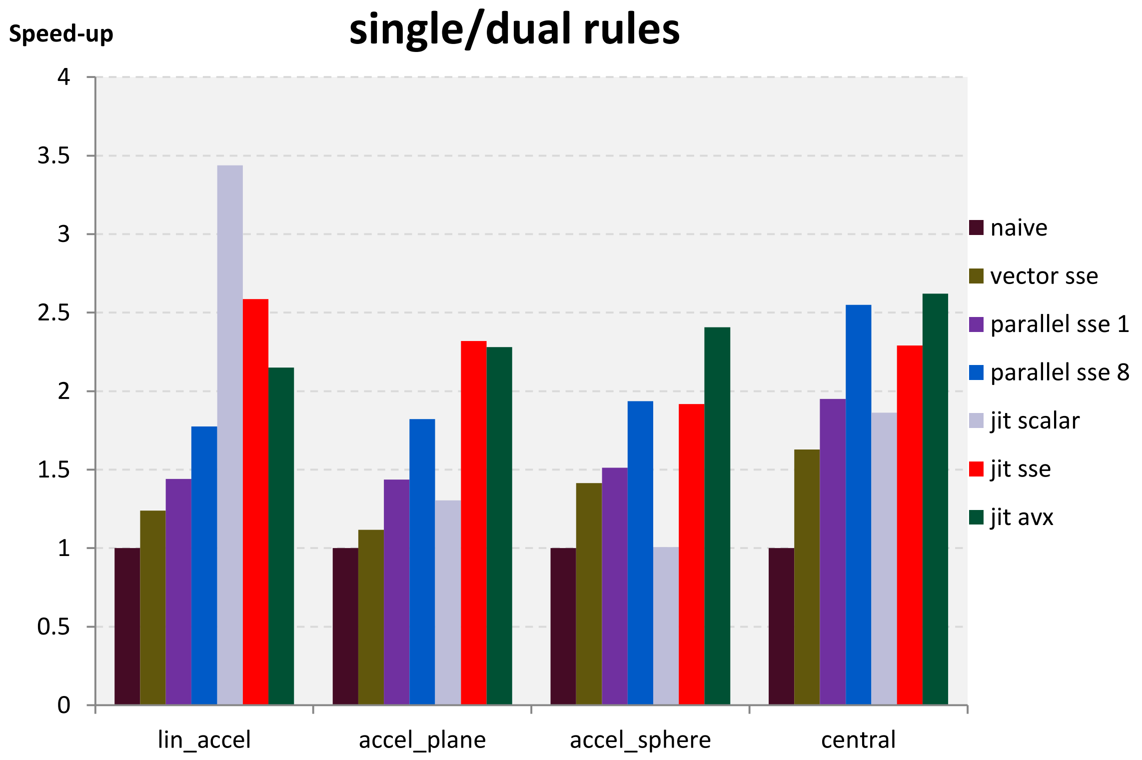
Results – JIT compiler in Detail

- **Compile time in sub millisecond range**
  - Code generation constant
  - Scheduling heavily depends on rule composition and number of instructions
  - Fast enough for real time usage



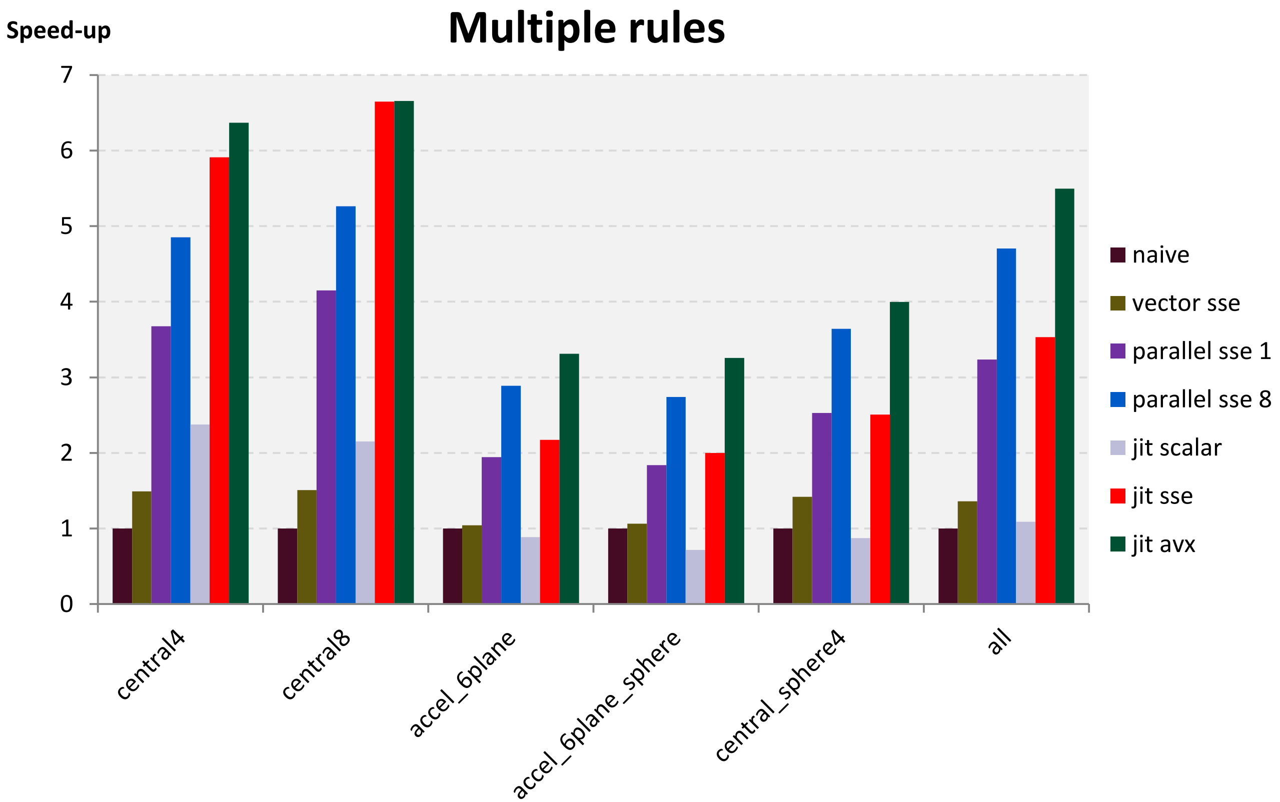
Results - Performance

- **SSE/AVX vectorization loses a lot of its potential gains due to data shuffling**
- **Speed-up of ~2x instead of theoretical 4x/8x**



Results - Performance

- Given enough work SSE/AVX vectorization gains outweight data shuffling losses
- Speed-up ~2.5/3.5x for SSE/AVX



Conclusions

- **Simple JIT compiler competitive/surpassing optimized code**
  - Excluding loop unrolling, which the JIT compiler cannot yet do
- **“messiness” of optimizations hidden from programmer**
- **JIT compiler can be improved using various known compiler techniques**

The End

&

Questions