A System for Runtime Loop Optimization in the Jikes RVM

The University of Manchester

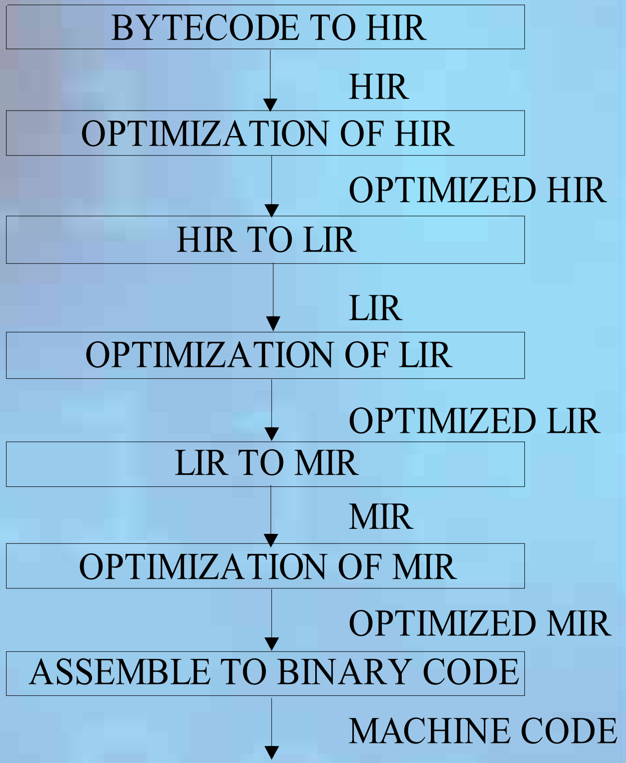
Jisheng Zhao

Background

- Jikes RVM Java Virtual Machine
  - A good test bed for evaluating research ideas
  - Flexible and modularized architecture
    - Runtime service (virtual processors, light-weight threads, etc.)
    - Optimizing compiler and Baseline compiler
    - Memory management (MMTK)
    - Written in Java (more than 90% of code)

Background

- Jikes RVM Optimizing compiler
  - 3 level optimization frameworks (Java code already compiled to bytecode)
  - A series of optimizing compilation phases in different intermediate representation(IR) levels
  - Simple loop optimization framework
  - Extended Array Static Single Assignment (SSA form)



BYTECODE TO HIR

HIR

OPTIMIZATION OF HIR

OPTIMIZED HIR

HIR TO LIR

LIR

OPTIMIZATION OF LIR

OPTIMIZED LIR

LIR TO MIR

MIR

OPTIMIZATION OF MIR

OPTIMIZED MIR

ASSEMBLE TO BINARY CODE

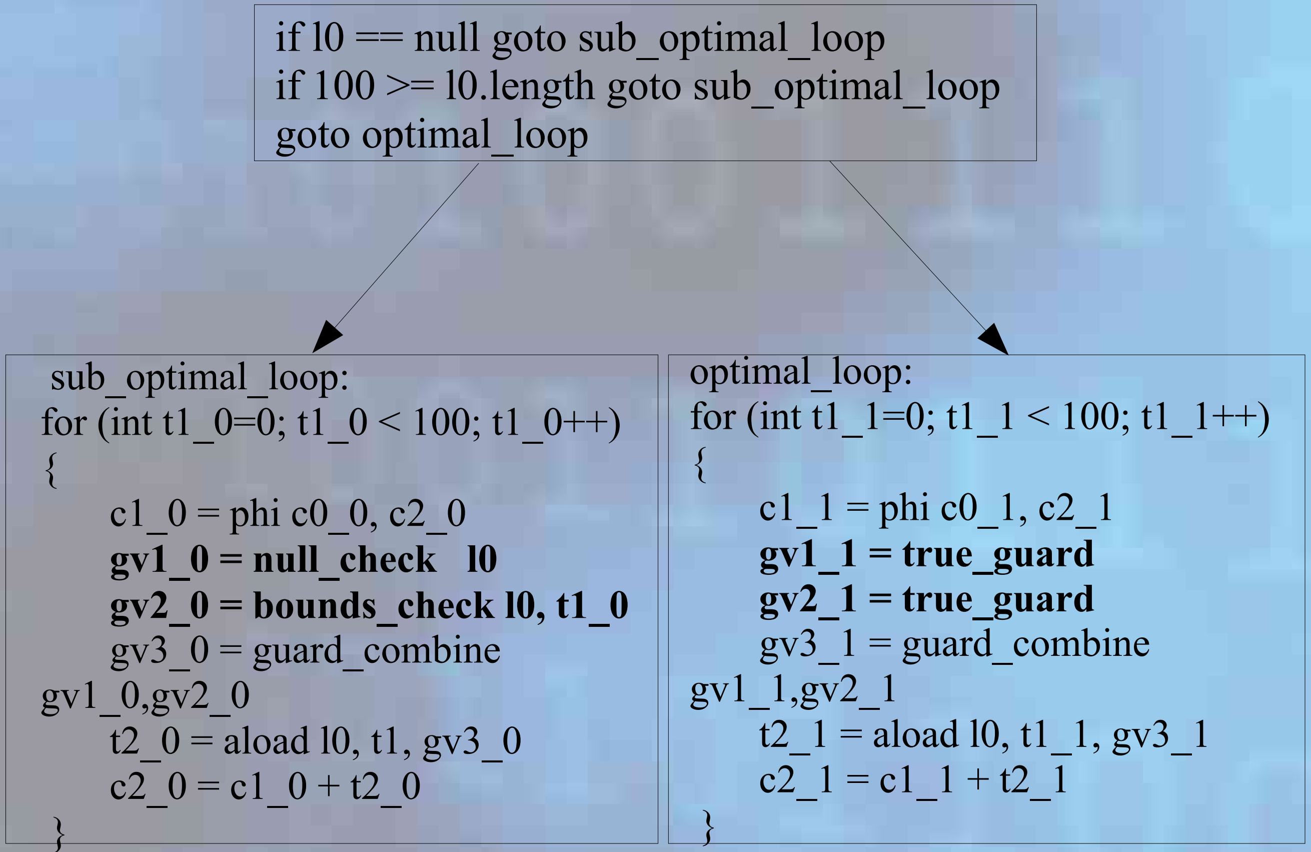
MACHINE CODE

Array Bound Check and Null Check Elimination

- Observation:
  - Array Bound Check on Demand (ABCD) limited effect
- Consider eliminating redundant checks using loops
  - Test bounds before executing more optimal loop
  - Run original loop if possible exception

```
for (int t1=0; t1 < 100; t1++) {
      c1 = phi c0, c2
      gv1 = null_check   l0
      gv2 = bounds_check l0, t1
      gv3 = guard_combine gv1,gv2
      t2 = aload l0, t1, gv3
      c2 = c1 + t2
}
```

Array Bound Check and Null Check Elimination



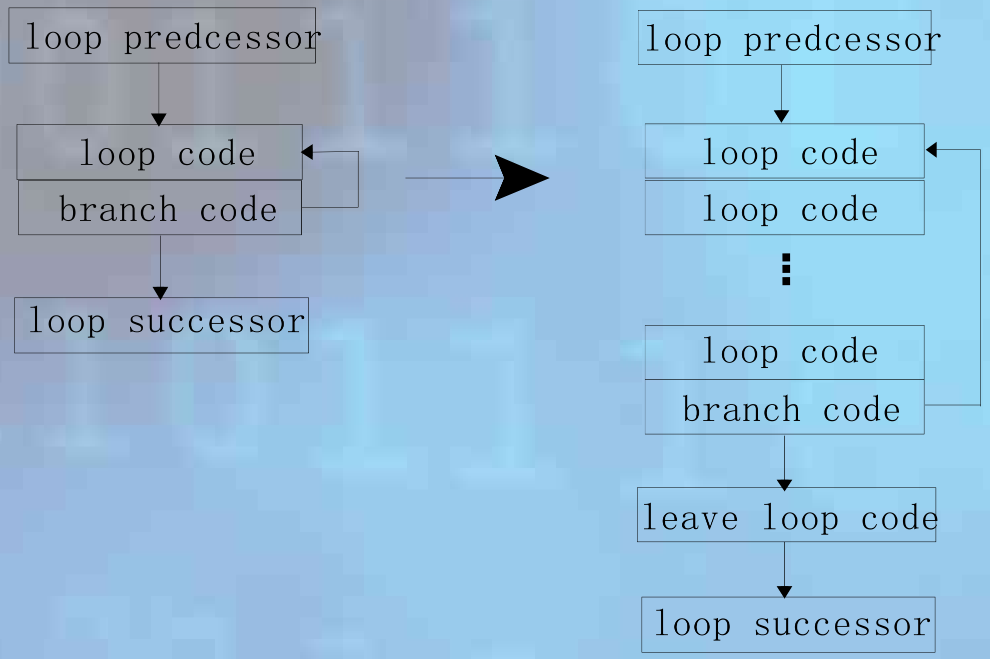
```
if l0 == null goto sub_optimal_loop
if 100 >= l0.length goto sub_optimal_loop
goto optimal_loop
```

```
 sub_optimal_loop:
for (int t1_0=0; t1_0 < 100; t1_0++)
{
     c1_0 = phi c0_0, c2_0
     gv1_0 = null_check   l0
     gv2_0 = bounds_check l0, t1_0
     gv3_0 = guard_combine
gv1_0,gv2_0
     t2_0 = aload l0, t1, gv3_0
     c2_0 = c1_0 + t2_0
 }
```

```
optimal_loop:
for (int t1_1=0; t1_1 < 100; t1_1++)
{
     c1_1 = phi c0_1, c2_1
     gv1_1 = true_guard
     gv2_1 = true_guard
     gv3_1 = guard_combine
gv1_1,gv2_1
     t2_1 = aload l0, t1_1, gv3_1
     c2_1 = c1_1 + t2_1
 }
```

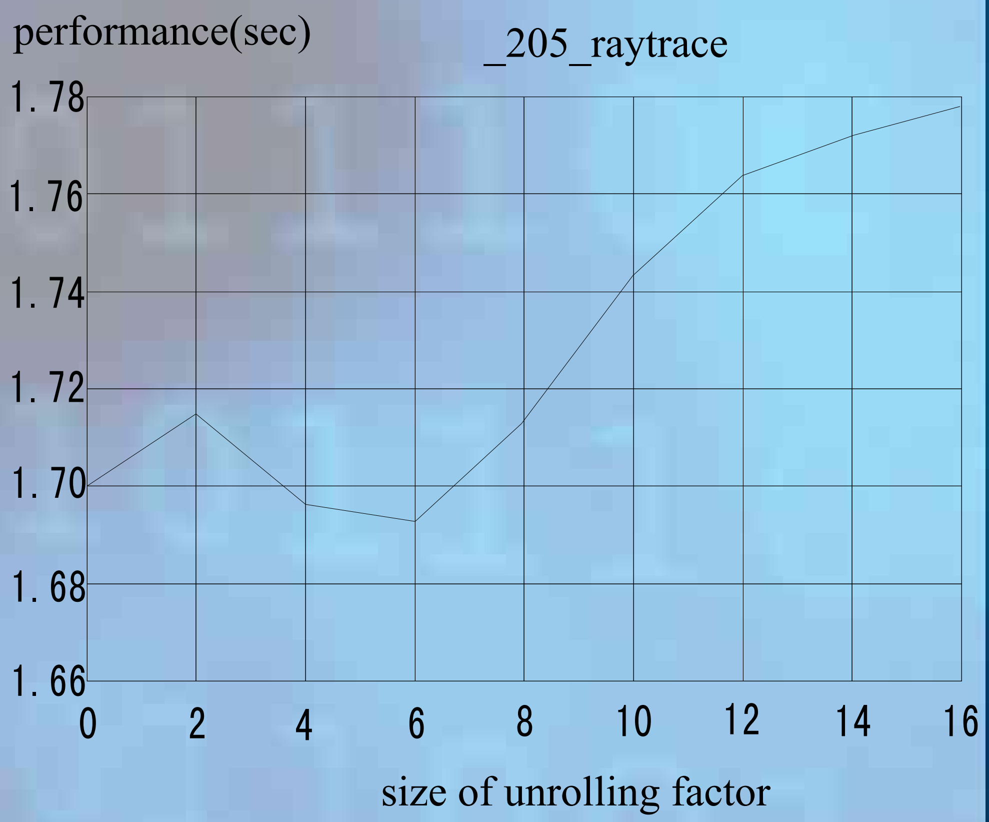
Constant Loop Unrolling

- Loop Unrolling
  - Constant number of iterations
  - Eliminate redundant branch code



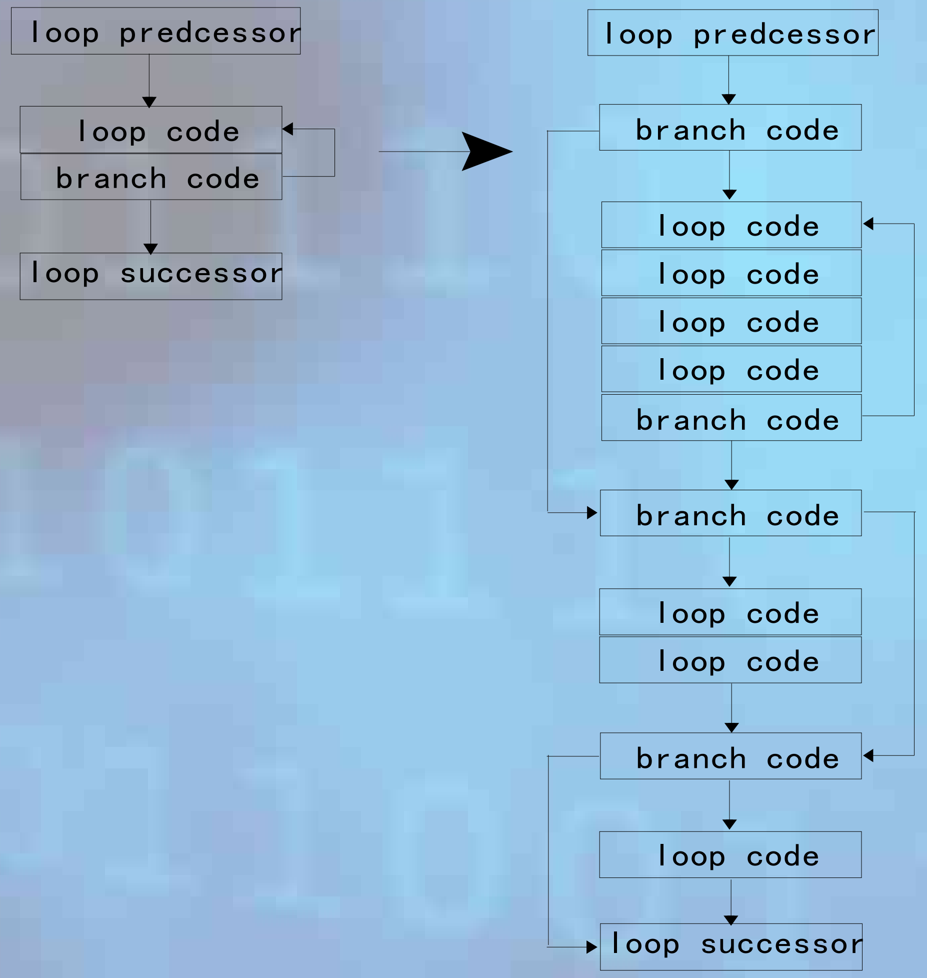
Constant Loop Unrolling

- The size of unrolling factor will affect the workload of dynamic compiler
  - More unrolled iterations increases number of basic blocks



Affine Loop Unrolling

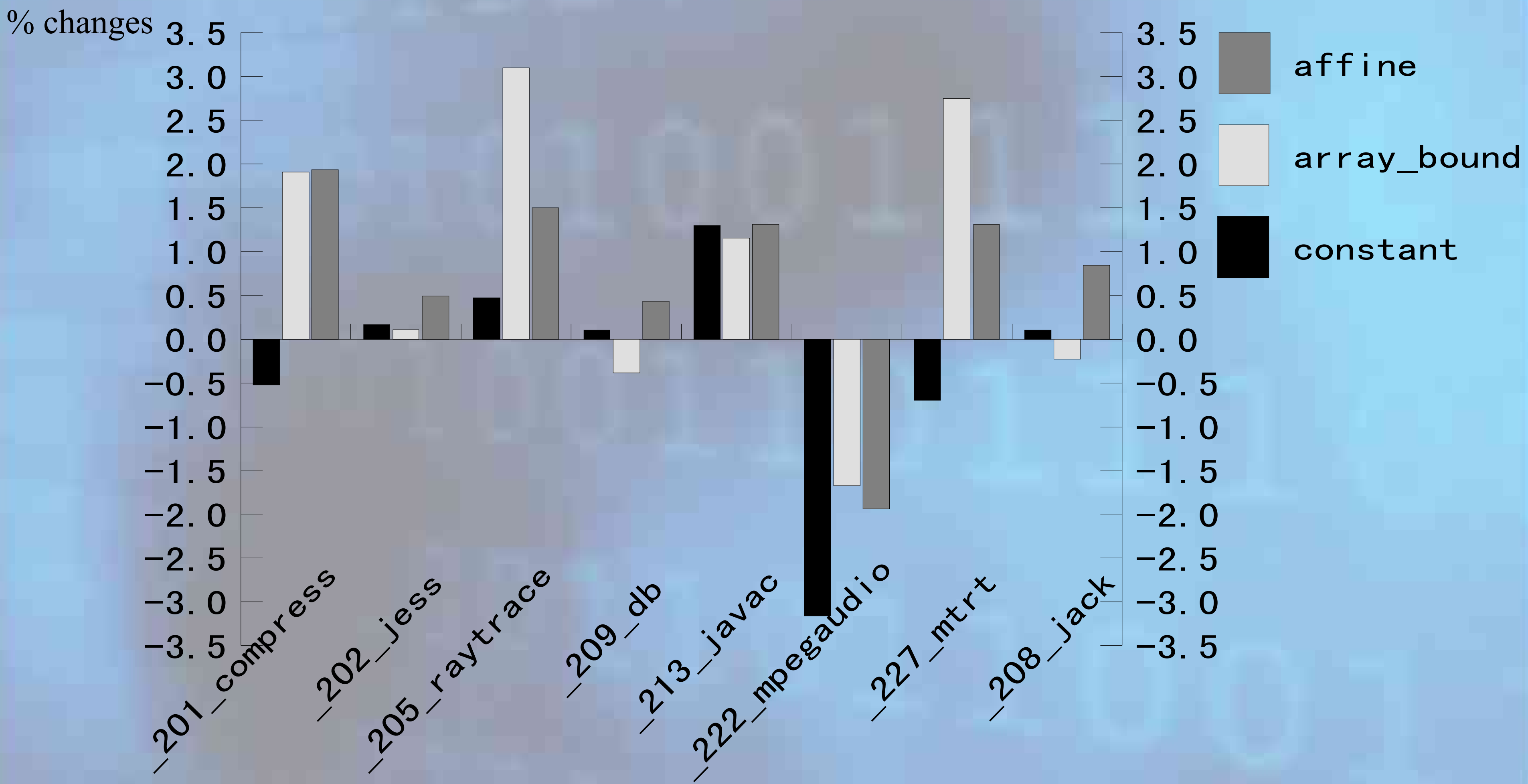
- A general loop unrolling strategy
  - General model for any number of iterations
  - Eliminate most of the redundant branch code



Affine Loop Unrolling

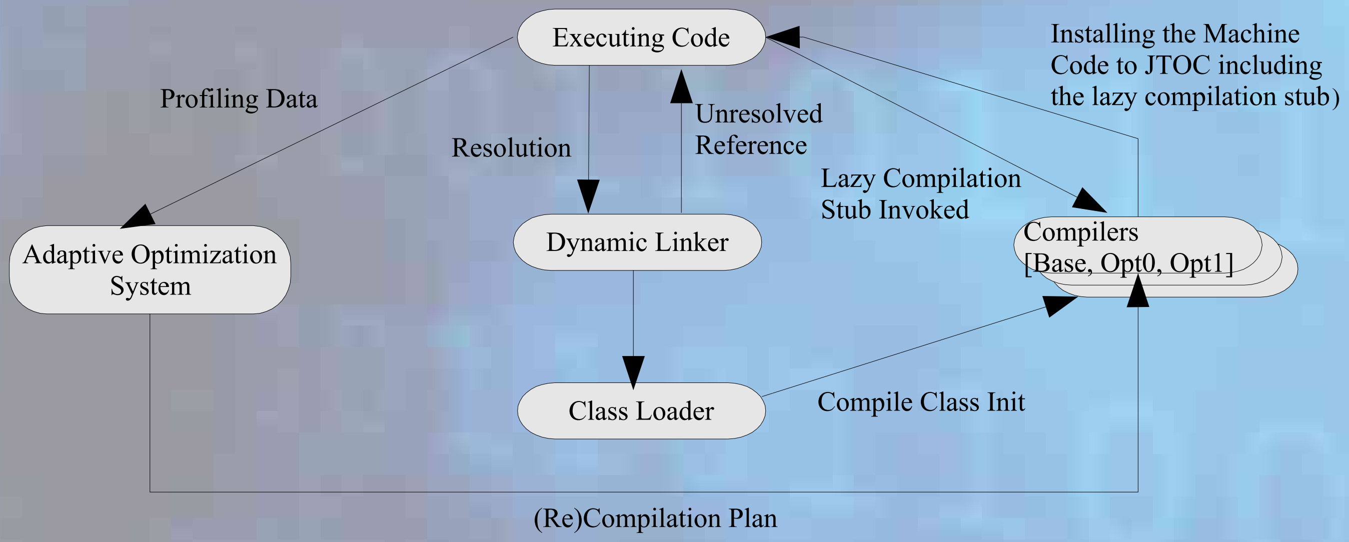
- Compared with the original loop
  - 4 blocks of branch code in this model
  - Number of iterations should be larger than 4
- Division factors (now, we use 4, 2, 1)
  - Other factors, eg. 8, 4, 2, 1
  - Increase work load for dynamic compiler (same problem as constant loop unrolling)

Experimental Result



Effect on Dynamic Compilation

- Adaptive optimizing compilation in Jikes RVM
- Trade-off between the cost of dynamic compilation and the benefit got from loop optimization


Executing Code
Installing the Machine Code to JTOC including the lazy compilation stub)
Profiling Data
Resolution
Unresolved Reference
Lazy Compilation Stub Invoked
Adaptive Optimization System
Dynamic Linker
Compilers [Base, Opt0, Opt1]
Class Loader
Compile Class Init
(Re)Compilation Plan

Future Work

- Chip Multi-Threaded (CMT)
- Chip Multi-Processor (Jamaica CMP)
  - Allows distributed execution of fine-grained parallel code sections
- Loop-Level Parallelization (LLP)
- Challenges
  - Modeling loops and heap based data dependences
  - Java exception semantics in parallel code

Questions