Interpretation – Interesting points made in the paper

- Flat sequence layout of operations vs graph layout
  - Flat sequence is easier to manipulate – fast
  - VM instructions

- “Level” of operations
  - Amount of interpreter work per amount of useful work
    - Impacts the difference in performance between the interpreter and the equivalent native code execution

- This work targets LOW LEVEL bytecodes
  - Those with high dispatch-to-work ratios (dispatch rate)
  - Note that the Python numbers presented earlier
    - Python has low dispatch rates, so interpreter overhead is in the noise
    - That is, these optimizations (that target the interpreter overhead) aren’t likely to have much impact
Interpretation – Interesting points made in the paper

• “Level” of operations
  ■ Amount of interpreter work per amount of useful work
    ‣ Impacts the difference in performance between the interpreter and the equivalent native code execution
      ◆ Large number of simple operations
        - Interpreters are slowest relative to native code execution
  ■ JVM vs GForth
    ‣ Dispatch-to-real-work ratio of GForth is higher (simpler VM instructions)
      ◆ JVM – fewer dispatches for same amount of work
    ‣ JVM: more time outside of interpreter loop (GC, verification)
    ‣ GForth caches topmost operand stack element in a register
    ‣ 16.5% of retired machine instructions are ind. branches (6.1% for JVM)
      ◆ Opts that reduce branch misses will benefit GForth more than JVM
Interpretation – Interesting points made in the paper

- Flat sequence layout of operations vs graph layout
  - Flat sequence is easier to manipulate – fast
  - VM instructions
- The biggest problem with interpretation on performance
  - Branch mispredictions
  - The deeper the pipeline the worse the cost
  - Again for bytecodes with high dispatch rates
    - And the overhead of the dispatch loop
      - Two sources of overhead: Number of dispatches, cost per dispatch
- Solutions: replication, superinstructions