Administration

- **Minijava project**
  - Phase 6 (Register allocation)
    - due Yesterday
    - There is a contest 😊
    - Link is on the course homepage
  - Overall
    - Due Wednesday April 11\textsuperscript{th} 11:59pm
    - A few more days to work out any issues
The Plan

• Dataflow analysis
• Optimization
• A Few final words on optimization
• Garbage collection
  -- A few words on techniques
  -- Integration with the compiler
• Other compiler-like things
• JIT
What does a compiler do?

- Translate a program from one language to another
- Preserve meaning
- Usually a high level programming language to a machine level language
- Can also be to another high level programming language
  - Source to source compilation
  - Original C++ compilation technique
Other Compiler-Like Things

• Other tools use similar techniques, tools, or data structures
  – Assemblers
  – Linkers
  – Performance analyzers
  – Debuggers
Assembler

• Takes assembler source (text) and produces relocatable machine code (binary)
  – Usually 2-pass
    • because forward references are allowed
  – Rather more simple than a compiler
    • Format is more regimented
    • See an instruction in text, emit it in binary
      – No type checking, no IR, no instruction selection, no register allocation
    • A few local optimizations
      – Choosing the shorter form of instructions
Linker

• Takes multiple relocatable binaries and creates a single executable
  – Each binary assumes it is going to be at location 0, but remembers all the places where that assumption was used
  – Linker matches up symbols
    • binary1 defines a symbol that binary2 uses
  – Linker deals with libraries
  – Emits a finished executable
    • Except for things that will be dynamically linked
Performance analyzer

- Why does my program take so long to run?
- Why does my program use so much memory?
- Instrument the code to record interesting events during execution
- Produce a report after execution is finished
- For example, for C:
  - gcc –pg
  - gprof
  - Instruments (on the MAC)
- Can be a separate tool or option to compiler
Debugger

- Debuggers help programmers understand how their programs execute
- A reverse mapping from what a compiler does:
  - Compiler maps from source to executable
  - Debugger maps from executable back to source
    - Lines of code
    - Data values
    - Data structures
    - Often have the ability to execute or interpret bits of the program
      - evaluate expressions
      - call functions/methods
Just in Time Compilation (JIT)

• Many language implementations don’t compile directly to machine code
  – mostly for portability reasons
    • Java

• Instead they compile to an IR (Java bytecodes in the case of Java)

• Which is then executed somehow
  – interpreted
  – compiled just before execution (just-in-time)
  – a combination of these two
To compile or not to compile

• Early implementations compiled each method just before it was executed for the first time
  – Hence the name, “just-in-time” compilation

• This is much better than compiling everything before executing anything
  – lower startup latency
  – some methods never execute (in a normal execution) so never need to be compiled

• But, still not great
  – Why?
Why not compile before first run?

• Many methods execute just once (or twice)
  – Examples?

• A very few methods consume the lion’s share of the processing time
  – This has been observed since at least Donald Knuth’s 1971 paper on FORTRAN programs
    “We also found that less than 4 per cent of a program generally accounts for more than half of its running time.”
So, then what?

- You (as the runtime) have choices:
  - Interpret
    - makes sense for methods that are executed so few times that compiling them isn’t worth it
  - Compile
    - makes sense for methods that are executed a lot
  - Interpret for a while, then compile
    - Nice compromise
  - Profile driven
    - Use feedback from the execution profile to decide
How to choose

- Schilling 2003
  - the size of the method (150)
  - the number of times it has been executed (40)
JIT Opportunities

• After you have executed a method a few times, you might know more about it than was discernable statically (before running it)
  – What is the common code path through the method (which branches are taken how often)?
  – What other methods does it invoke?
  – What (actual) types do the manipulated objects have?
  – What parameter values are actually passed in?
What is slow?

- Optimizing (for run time) requires knowledge about what things are fast and what things are slow
- What things are fast on today’s processors?
- What things are slow on today’s processors?
What is slow?

- Optimizing (for run time) requires knowledge about what things are fast and what things are slow

- What things are fast on today’s processors?
  - register operations
  - memory operations that hit in the cache

- What things are slow on today’s processors?
  - memory operations in general
    - in particular, indirections through memory
    - like virtual function invocations that go through the virtual function table
  - branches
    - branch prediction isn’t perfect
Speculative compilation

• Assume certain things will be true in the future
• Generate code and optimize based on these assumptions
• Also generate code that checks the assumptions
• When an assumption is violated – re-optimize
What kind of assumptions?

- Variables will have particular values
  - final fields won’t change value after they are initialized
  - many other fields may, but don’t in practice
  - parameters are often the same from one call to another

- The class of an object being manipulated won’t change

```java
Set<Integer> t;

t = ...

if (t.contains(e))
...
```
When a variable doesn’t change value ...

- You can treat it as a constant
  - This technique used to be known as partial evaluation
- Once you know that some values are constant, create a new version of the program that takes advantage of those constant values
  - Sizes of things
  - Debug flags
  - Localization settings
When the class of an object doesn’t change ...

- You don’t have to search for the method to invoke
  - It will be the same one that you invoked last time through the code
    - Inline cache
  - Sometimes there will be more than one class (but rarely will there be “lots” of classes)
    - Polymorphic inline cache
- In the JIT context, you can also expand the body of the method in place of its invocation
  - Invoking many small methods (getters and setters among others) has high overhead
  - Inline expansion removes this overhead, as well as giving other optimizations bigger methods to work on
How to invalidate when changes happen?

- Whenever the JIT takes advantage of something that appears to be constant, it must make sure to detect when that “constant” changes
  - New classes can be loaded dynamically
  - The computation may enter a “phase change” where it starts doing very different things
- The JIT must install “triggers” that will fire when “constants” change
  - On assignments to “constant” variables
  - On loading new classes
  - On hitting a new class where inline expansion was performed
- When a trigger fires, discard and re-optimize