6.110 Computer Language Engineering

Recitation 8: Phase 5 infosession

April 19, 2024

Weekly updates ←

Phase 5 info

Wrapping up phase 4…

- **Project phase 4 is due today 11:59PM!!!**
  - **This includes the report!**
  - Remember to add your teammates to the submission!
  - If you need last-minute help, please come to OH today from 2-7pm.

New releases

- Project phase 5 has been released, due **Monday, May 13**
- Phase 4 LLM questionnaire due **Monday, April 22**
  - Counts as extra credit towards participation grade
- Weekly Check-in and mini-quiz are due **Thursday, April 25**
  - We promise to actually release a mini-quiz by this weekend
  - Reminder: these are graded on completion – please submit!!

Quiz 2: **Friday, May 3**

- Quiz will be in class, worth **10%** of the overall grade
- Covers lecture content after Quiz 1:
  - Program analysis
  - Foundations of dataflow
  - Register allocation
  - Loop optimizations
  - Parallelization
- Past quizzes are on course website
- Quiz review session on **Wednesday, May 1** during re-lecture time (4-6pm in 26-322).

Weekly plan: **Week N+1**

Mon 4/22	**Tue 4/23**	**Wed 4/24**	**Thu 4/25**	**Fri 4/26**
No lectures for the rest of the semester				**Recitation** TBD
Due: Phase 4 LLM questionnaire		**Re-lecture** Optimizations	**Due:** Mini-quiz, weekly check-in	

Weekly plan: **Week N+2**

Mon 4/29	**Tue 4/30**	**Wed 5/1**	**Thu 5/2**	**Fri 5/3**
No lectures for the rest of the semester				**Quiz 2** Dataflow and optimizations
		Quiz review (4-6pm in 26-322)	**Due:** Weekly check-in	

Weekly updates

Phase 5 info ←

Project overview

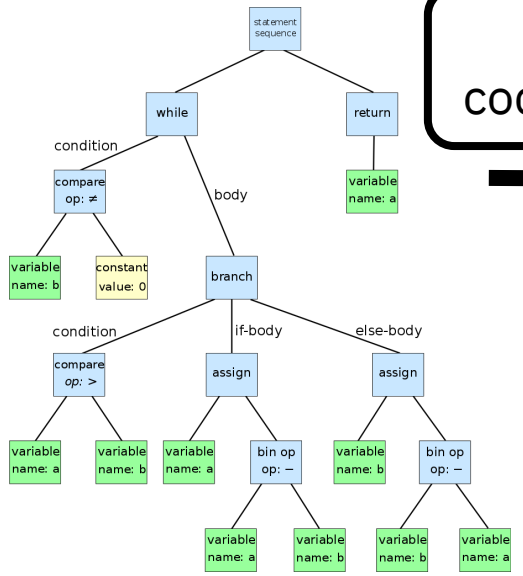
```
import printf;

void main() {
…
```

Decaf source file

Phase 1. Does it have the right structure? (syntax)

Phase 2. Does it make sense? (semantics)



Internal representation

Phase 3
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

Even more optimized
x86-64 assembly

Phase 5. How can we make the output code faster?

Phase 4. What can we learn about the program? (dataflow analysis)

Again, we'll require one specific optimization, but other than that you are free to implement whatever your heart desire.

On the last day of class (May 14), there will be a **compiler derby** to find which team's compiler produces the fastest code!

Logistics and requirements

Phase 5 overview

- Generate correct code that is as fast as possible!
- **Required:** implement **register allocation**
  - You can choose between **graph coloring** or **linear scan**
- **Optional:** everything else!
  - More dataflow optimizations
  - Peephole optimizations
  - Instruction selection and scheduling
  - Parallelization (e.g. data parallelization, SIMD)

Notes on performance testing

- Test cases from previous phases are too simple to be used as effective benchmarks.
- We'll run your compiler on a benchmark suite of more complex programs simulating real-world workloads.
  - Some of these will be released; others will only be available on the autograder

Notes on performance testing

- We **strongly recommend** setting up a benchmarking framework locally.
  - This should provide a much tighter feedback loop than using the autograder.
  - Use provided benchmarks + your own benchmarks
    - Some provided tests require linking with the provided library in `derby/lib/`
  - Recommendation: use Hyperfine for simple timing, use tools like gprof or perf for profiling

Phase 5 autograder

- The autograder for phase 5 will work a bit differently from previous phases.
- We have a dedicated server that will execute submissions in a round-robin fashion, one at a time. We expect performance variations to be less than 5%
- CPU information is available on the course website (https://6110-sp24.github.io/phase-5#phase-5-autograder-information)

Submission

- Phase 5 is due **Monday, May 13.**
- As usual, two items to be submitted on Gradescope
  - Design document
  - Code submission
- Submission is both for Phase 5 and for the whole compiler project

Grading

- **10%:** required Phase 5 tasks:
  - Register allocation
  - Design document – explains choice of optimizations in **`-O all`** and design of register allocator
- **30%:** overall compiler project
  - Remaining parts of design document (5%)
  - Correctness (15%), based on previous test cases + benchmark suite
  - Performance on Derby benchmark suite (10%)
    - We'll have pre-set cutoffs

Specifications

- Your compiler should be **correct** with or without optimizations
- When running
  `./run.sh <filename> –t assembly`
  on a valid input file:
  - Outputs x86-64 assembly code to the output file (or stdout if `–o` is not specified)
- We'll assemble using
  `gcc -O0 -no-pie output.s -o output.exe`

CLI for optimizations

Same as phase 4!

- `-O cse` turns on common subexpression elimination only
- `-O regalloc` turns on register allocation only
- `-O cse,regalloc` turns on common subexpression elimination and register allocation only
- `-O all` **turns on "all optimizations"** (we'll test performance with this option)
  - This means "put all your best effort at producing the fastest code"
  - You decide which optimizations are actually run, in which order, and how many times

Design document

- Explains technical details **for the whole project** (you can reuse portions of previous design documents)
- Includes the following sections:
  1. Design *(for Phase 5: discussion of "all optimizations" option and each optimization you implemented)*
  2. Extras
  3. Difficulties
  4. Contribution
- If you used LLMs, also describe how you used them and provide chat logs

1. Design

- Overview of your design, including design choices you made and design alternatives you considered.
- This section should help us understand your code
- For Phase 5, please include:
  - A detailed discussion of your **`-O all`** option, including which optimizations are performed, which order they are performed in, how many times they are performed, and how you arrived at this choice
  - Details of each optimization you implemented (more info on next slide)

1. Design — details

- For each optimization you implemented, please include:
  - a brief explanation of how your optimization worked (this should convince your reader that the optimization was beneficial, general, and correct)
  - a sample test case, with generated code before and after, included under `doc/phase5-code/` in your repository
  - if possible, include empirical evidence that proves the effectiveness of your optimization.

Other sections

2. Extras:
   - Any clarifications, assumptions, or additions you made
   - Any interesting debugging techniques, build scripts
   - Anything cool you'd like to share!
3. Difficulties:
   - List of known problems with your project, and as much as you know about the cause
   - Any issues from previous phases that you fixed
4. Contributions: A brief description of how your group divided the work

Some optimizations

plus some comments

Graph-coloring regalloc

(i.e. regalloc taught in lecture)

1. Compute webs based on def-use chains
2. Compute interference graph between webs
3. Assign registers by coloring interference graph, and spill if necessary
   - Heuristics for spilling can be important!

See also Ch. 16 of Whale book / Ch 9.7 of Dragon book

We suggest this if you want the best performance

Linear scan regalloc

Assumes a linear representation of IR instead of a CFG structure (i.e. give every statement/block a unique integer id and order your IR by that id)

- Compute live intervals instead of webs
- Makes register allocation much simpler and faster at compile-time, but is worse at run-time

See paper: https://dl.acm.org/citation.cfm?id=330250

We suggest this if you want the simplest option

Phase 4 dataflow optimizations

All three optimizations work together really well, especially if you can deal with array elements

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

Constant optimizations

- **Constant propagation:** propagate values of constants at compile time
- **Constant folding:** evaluate constant expressions at compile time
- **Algebraic simplification:** simplify expressions like a + 0, a * 1, etc..

These are relatively simple to implement and pretty helpful

Peephole optimizations

Basically “find-and-replace” on emitted instructions + instruction selection. Examples:

- Replace **x / 8** by **x >> 3**
- Remove jumps to the label immediately after
- Remove push immediately followed by pop to same register

Resources: Agner Fog’s instruction tables, https://uica.uops.info

This is easy to implement and pretty helpful!

Loop optimizations

Loop-invariant code hoisting: move code that always compute the same thing out of the loop

- Can be helpful if the thing that's being recomputed again and again is expensive, but can also lead to larger webs in regalloc

Induction variable optimizations

- Best for array accesses, generally not the most helpful for Decaf

Function inlining

Replace a call to a function f by the body of f.

- Can help with dataflow + register allocation (can now reason about variables across the function call boundary)
- Also helps with tail recursion

Instruction scheduling

Reorder instructions to reduce immediate dependencies

- Recall 6.004: we want to use many stages of the pipeline at once! If there's a data dependency then we can't do this

Resources: Whale book Ch. 17, Agner Fog's instruction tables, https://uica.uops.info

Parallelization

Do multiple operations in parallel, e.g.

- **SIMD:** emit vector instructions that operate on multiple pieces of data at once
- **Data parallelization** across iterations of a loop

⚠️ **Parallelization is very tricky to get right. If you want to try this, please talk to the TAs.** ⚠️

Words of advice

- **Start simple!**
  - Start with very simple test cases so that you understand what's happening
- **Test often, both for correctness and performance**
  - An optimization needs to be correct, and you should make sure it's actually making the generated code faster.

- **Think about what optimizations to implement, and also *how to implement each optimization most effectively***
  - Some optimizations go well together, e.g. common subexpression elimination + dead code elimination
  - Improving some optimizations (e.g. add dataflow for array elements) might be more worth it than trying to implement a whole new optimizations
  - Freeing another register for your regalloc to freely use can also be surprisingly helpful

- **Focus on the slowest parts first**
  - A program can’t be faster than its slowest part. It’s much more worth it to optimize the slow part than the part that’s already fast.

- **The little things also matter**
  - You will likely find a lot of small inefficiencies in your generated code. They add up.