

EECS483 D12: Project 5 Overview

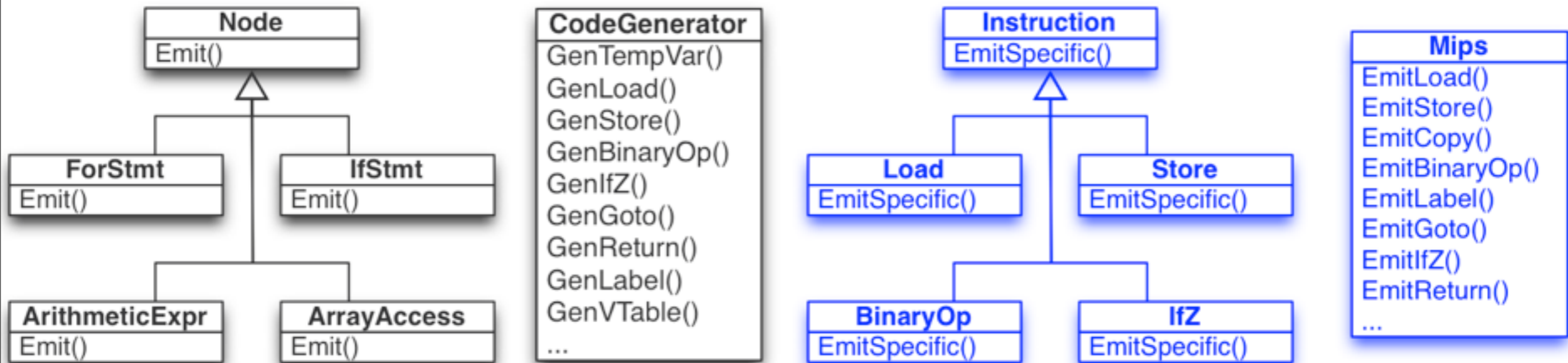
Chun-Hung Hsiao

April 5, 2013

Announcement

- Project 5 released on CTools
 - Due on 4/22/2013
- We will manually check your LAST submission
 - Provide a README file to
 - Briefly outlining your implementation
 - Explain your basic design decisions
 - Summarize experimental results noting which files benefited from your pass
 - Write sufficient comments in your code to help us read your code

Project 5 Overview



Project 5: Register Allocation (1/3)

- The original register allocation is awful!
 - If the following code is given
 - $a = 1;$
 - $b = 2;$
 - $c = a + b;$
 - The default TAC-to-MIPS conversion is
 - $\$t2 = 1;$
 $*(\text{addr of Location } a) = \$t2;$
 - $\$t2 = 2;$
 $*(\text{addr of Location } b) = \$t2;$
 - $\$t0 = *(\text{addr of Location } a)$
 $\$t1 = *(\text{addr of Location } b)$
 $\$t2 = \$t0 + \$t1$
 $*(\text{addr of Location } c) = \$t2$
 - Only 3 registers are used!

Project 5: Register Allocation (2/3)

- We can have better register usage
 - If the following code is given
 - $a = 1;$
 - $b = 2;$
 - $c = a + b;$
 - Map each Location to a register
 - Location a: \$s0
 - Location b: \$s3
 - Location c: \$t4
 - Then generate efficient code
 - $\$s0 = 1;$
 - $\$s3 = 2;$
 - $\$t4 = \$s0 + \$s3$

Project 5: Register Allocation (3/3)

- Compute interfering locations and build the interference graph
 - Liveness analysis at IR/TAC level
- Allocate registers for each location via graph-coloring
 - Map locations to MIPS registers
- Generate MIPS instructions based on the allocation
 - If a location is mapped to a register, use it directly
 - If a location is not mapped, use a free register to load the value before use or store the value after def
 - What if all registers are used?

Step 1: Identifying Basic Blocks

- The following TAC starts a basic block:
 - BeginFunc
 - Label
 - Any instruction after a Goto, IfZ or Return
 - In fact any instruction after a Return is dead
- The following TAC ends a basic block:
 - Goto
 - IfZ
 - Return
 - EndFunc
 - Any instruction before a Label

Practice: Allocating Temporary Variables

- All temporary variables are only assigned once
 - This is called Static Single Assignment (SSA) form
- If no optimization applied, they are only used in the same basic block
- The simplest register allocation is just doing a local liveness analysis and allocate register for temporary variables!
 - You can start your PP5 with this simple allocation as a practice
 - You are still required to implement the whole global register allocation though

Step 2: Construct Intra-procedural CFG

- The “global” analysis described in class is just a intra-procedural analysis
 - Analysis within a single function
 - CFG of a function is static and can be analyzed easily
 - Inter-procedural control flow is dynamic due to function calls and thus much harder to analyze
- One pass to find the label-to-basic-block mapping
- Another pass to construct the CFG of each function
 - Examine each Goto and IfZ to find the predecessors and successors of each basic block
 - Introduce a pseudo exit node for each function

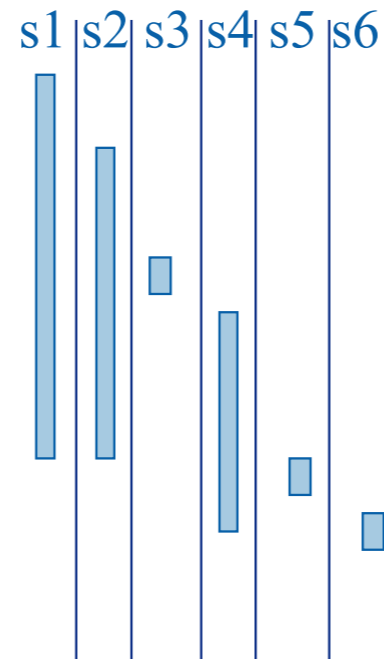
Step 3: Liveness Analysis

- For each instruction, find live variables *before* executing it
- Consider the following piece of code annotated with live variables
 - {a, b, e} $c = a + b;$
 - {c, e} $d = c * e;$
 - {d} $\text{return } d;$
- Locations a, b and e lives before the first instruction so we need to allocate them into different registers
 - So do c and e
 - c can reuse registers that have been used by a or b

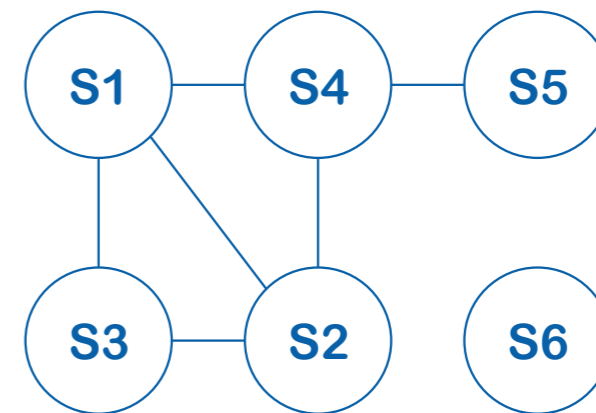
Another Approach: Live Range Analysis

```
s1 ← 2
s2 ← 4
s3 ← s1 + s2
s4 ← s1 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

Live Range



Interference



- Instead of keeping track of live variables, we keep track which definition is alive at each instruction
- Variables that use the name names but have different definitions are treated as distinct variables
- Useful if you want to reschedule the instructions

Step 4: Register Allocation

- Use all available registers and apply the graph-coloring algorithm
- Choose locations to spill if there are not enough registers
 - Locations with lower spill cost are better
 - Locations that interfere with more locations are better
 - Heuristic: pick the one that has the smallest value of (spill cost / #neighbors)
 - More complex strategies exist
- When spilling, you need more registers!
 - Keep some special registers for spilling
 - Or do iterative spilling

MIPS Registers

- MIPS architecture contains the following registers
 - \$0: zero, read-only -- cannot use
 - \$at: assembler temporary -- reserved
 - \$v0 - \$v1: function result registers
 - \$v0: returned value -- save it before function calls
 - \$a0 - \$a3: function argument registers
 - \$a0 - \$a2: used in built-ins -- save them before calling
 - \$t0 - \$t9: temporary registers
 - \$s0 - \$s8: saved registers
 - \$k0 - \$k1: kernel registers -- reserved
 - \$gp: global pointer -- reserved
 - \$fp: frame pointer -- reserved
 - \$sp: stack pointer -- reserved

Step 5: Use Allocated Registers

- Modify the MIPS emitting functions to emit assembly code based on our allocation
- For spilled locations
 - If it is a source, remember to load the value to a free register
 - If it is a target, remember to store the value from the result register
 - MIPS is 3-address, so we need at most 2 free registers if both source operands are spilled

Finished?

- Not really....
- Need to deal with function calls!

Function Entry

- For each allocated parameter, load them into their registers
- In the liveness analysis, the function entry should “kill” all parameter locations
- Remember to load spilled global registers as well

Function Call

- In the CFG, we do not keep track of the control flow of function call!
- Values of the registers in the caller may be destroyed by the callee!
 - Need to preserve the values
- Two strategies (calling convention)
 - Caller-saved registers - temporary registers
 - Callee-saved registers - saved registers
 - Different strategies incur different spill cost!
- You can choose whatever strategy you want

Example of Caller-/Callee-saved Registers

```
_f:
  BeginFunc 4
  _tmp1 = *($fp + 4);
  _tmp1 = _tmp1 * _tmp1;
  Return _tmp1;
EndFunc

_g:
  BeginFunc;
  _tmp2 = 1;
  _tmp3 = 2;
  _tmp4 = 3;
  _tmp5 = 4;
  _tmp6 = 5;
  PushParam _tmp6;
  _tmp7 = LCall _f;

  _tmp7 = _tmp7 * _tmp6;
  _tmp7 = _tmp7 * _tmp5;
  _tmp7 = _tmp7 * _tmp4;
  _tmp7 = _tmp7 * _tmp3;
  _tmp7 = _tmp7 * _tmp2;
  Return _tmp7;
EndFunc;

main:
  BeginFunc;
  _tmp8 = 6;
  _tmp9 = LCall _g;
  _tmp9 = _tmp8 * _tmp9;
  PushParam _tmp9;
  LCall _PrintInt;
  EndFunc;
```

- Should we use caller- or callee-saved registers for _tmp1?
- Should we use caller- or callee-saved registers for _tmp2 - _tmp6?
- Should we use caller- or callee-saved registers for _tmp8?

More Optimizations (Optional)

- You can choose to implement more optimizations
 - Dead code elimination
 - Constant folding
 - Common subexpression elimination
 - Constant propagation
 - Forward copy propagation
- Apply these optimizations before register allocation
- Design your analysis framework so you can use the same code skeleton to do all kinds of analysis
 - Just need to provide the transfer function and meet operator!

Thank you &
Good luck to your last project!
