Memory Hierarchy, Register Allocation, Linking, and Loading

Lecture 19

RIGs help with Register Allocation
Friday Discussions

- How are these going?
- Attendance is low
Automatic Memory Management

- We use an automatic memory management system *deallocates* objects when they are no longer used and reclaims their storage space.
- We must be *conservative* and only free objects that will *not be used later*
- **Garbage collection** scans the heap from a set of *roots* to find *reachable* objects. We will cover basic GC algorithms: *Mark and Sweep* and *Stop and Copy*
- We also use *Reference counting*, which stores the *number of pointers to an object* and frees it when that count reaches zero.
Notes on Garbage Collection

- The *garbage collector* is code that reclaims memory *at runtime*
  - i.e., a compiler will emit garbage collection code in addition to the program code
  - Some garbage collection libraries exist
    - [http://www.hboehm.info/gc/](http://www.hboehm.info/gc/) for C++

- The OS gives your process space in which to execute—your process must manage it appropriately
  - *memory leaks* are a result of *bad memory management*
  - Memory management is an active and important part of program analysis and compilers
Types of Garbage Collection

- Basic idea: starting at a set of roots (pointers on the stack), follow pointers to establish which objects are reachable from those roots
  - Unreachable items are garbage
  - This analysis is generally undecideable, so we err on the side of being conservative
    - Why is this undecideable?
Garbage collection Example

1 class Main {
  my_object : A;
  main () : Object {
    let w : A, x : A, y : A in {
      w <- new A;
      x <- new A;
      foo();
      y <- my_object;
    }
  }
  foo () : Object {
    my_object <- new A;
  }
};

15 class A { p : B; };
16 class B { z : A <- new A; };
Garbage collection Example (2)

1 class Main {
2     main () : Object { foo() };  
3     foo () : Object {
4         let x : A in {
5             x <- new A;  
6             bar ();
7         }
8     
9         bar () : Object {
10            let y : A in y <- new A
11         
12     }
13     
14     
15 class A { p : B; };  
16 class B { z : A <- new A; };
THE HARD DISK
YOU'VE BEEN WAITING FOR

$3398
10MB

MORE SOFTWARE
Included with the system is software for testing, for-
mating, I/O drivers for CP/M, plus an automatic
CP/M driver attach program. Support software and
drivers for MP/M and Cassette are also available. The
subsystem is now

MORE STORAGE
MORE SPEED
MORE VALUE
MORE SUPPORT

*company name*

XCOMP® introduces a complete micro-size disk

2 year warranty on parts and
Memory Hierarchy

- Computers are sequential digital circuits that require a mechanism for storing state.
- The simplistic Turing Machine uses a hypothetical tape to store arbitrary amounts of data.
- For practical performance, computers make use of hierarchical memory to track state:
  - Registers are super fast circuits, but low capacity (words).
  - CPU caches are very fast SRAM cells that supply larger capacity (10s of MB).
  - System memory is moderately fast DRAM cells that have a significantly larger capacity (100s of GB).
  - Hard disks are very slow, very large capacity storage (TB).
Register Allocation

Why can’t I hold

All these values in registers?
Why do we use registers?

- Variables usually refer to memory...
- &x yields a memory location
- We usually load variables into registers to execute on them
  1. Load from memory into registers
  2. Perform operation on registers
  3. Save results from registers to memory
Naïve Code

- Find a memory location for each symbol in scope
- For every TAC instruction
  1. Load operands (up to 3) into registers r1, r2, r3
  2. Emit ASM instruction
  3. Save result back to memory
Naïve Code

- Find a memory location for each symbol in scope
- For every TAC instruction
  1. Load operands (up to 3) into registers r1, r2, r3
  2. Emit ASM instruction
  3. Save result back to memory
- Correct and conservative approach, but...
Naïve Code

\[ x \leftarrow + y z \]
\[ w \leftarrow + z x \]
\[ u \leftarrow + z y \]

\[ l_a \ r_1 \ y \]
\[ l_a \ r_2 \ z \]
\[ a_d d r_3 \ r_1 \ r_2 \]
\[ s_t \ r_3 \ x \]
\[ l_a \ r_1 \ z \]
\[ l_a \ r_2 \ x \]
\[ a_d d r_3 \ r_1 \ r_2 \]
\[ s_t \ r_3 \ w \]
\[ l_a \ r_1 \ z \]
\[ l_a \ r_2 \ y \]
\[ a_d d r_3 \ r_1 \ r_2 \]
Register Allocation

- Naïve approach is correct
- But your code will be disgustingly slow
  - Something something memory hierarchy...
Register Allocation Intuition

- Fill as many registers as possible for as long as possible
- They’re super fast, and we hate going to memory
Register Allocation Intuition

- Fill as many registers as possible for as long as possible
- They’re super fast, and we hate going to memory

- If I want to run \( x \leftarrow + y \ z \)
  I will emit something like `add r1 r2 r3`
  - Different variables must reside in different registers during one instruction...

- **Idea** If variables are *live* at the same time, they must be loaded into separate registers
Register Allocation Intuition

- Fill as many registers as possible for as long as possible
- They’re super fast, and we hate going to memory

- If I want to run \( x \leftarrow + y z \)
  I will emit something like \( \text{add } r1 \ r2 \ r3 \)
  - Different variables must reside in different registers during one instruction...

- **Idea** If variables are *live* at the same time, they must be loaded into separate registers

- **Liveness?**
  - “Thank you for going over liveness analysis.”
“Liveness ranges”

- Knowing how long a variable stays alive is important
- A variable’s *Liveness range* is how long it stays alive
  - Where is each variable alive?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( t_0 \leftarrow \text{int 1} )</td>
<td>( {} )</td>
</tr>
<tr>
<td>2 ( t_1 \leftarrow \text{int 2} )</td>
<td>( {t_0} )</td>
</tr>
<tr>
<td>3 ( t_2 \leftarrow + t_1 t_0 )</td>
<td>( {t_1, t_0} )</td>
</tr>
<tr>
<td>4 ( t_3 \leftarrow * t_2 t_0 )</td>
<td>( {t_2, t_1, t_0} )</td>
</tr>
<tr>
<td>5 ( t_4 \leftarrow - t_3 t_2 )</td>
<td>( {t_2, t_1, t_3} )</td>
</tr>
<tr>
<td>6 ( t_3 \leftarrow + t_1 t_3 )</td>
<td>( {t_1, t_3} )</td>
</tr>
<tr>
<td>7 ( t_5 \leftarrow + t_3 t_3 )</td>
<td>( {t_3} )</td>
</tr>
</tbody>
</table>
“Liveness ranges”

- Knowing how long a variable stays alive is important
- A variable’s *Liveness range* is how long it stays alive
  - Where is each variable alive?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 ← int 1</td>
<td>{}</td>
</tr>
<tr>
<td>t1 ← int 2</td>
<td>{t0}</td>
</tr>
<tr>
<td>t2 ← + t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>t3 ← * t2 t0</td>
<td>{t2, t1, t0}</td>
</tr>
<tr>
<td>t4 ← - t3 t2</td>
<td>{t2, t1, t3}</td>
</tr>
<tr>
<td>t3 ← + t1 t3</td>
<td>{t1, t3}</td>
</tr>
<tr>
<td>t5 ← + t3 t3</td>
<td>{t3}</td>
</tr>
</tbody>
</table>

Liveness range of t1: \{3,4,5,6\}
“Liveness ranges”

- Knowing how long a variable stays alive is important
- A variable’s *Liveness range* is how long it stays alive
  - Where is each variable alive?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 ← int 1</td>
<td>{}</td>
</tr>
<tr>
<td>t1 ← int 2</td>
<td>{t0}</td>
</tr>
<tr>
<td>t2 ← + t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>t3 ← * t2 t0</td>
<td>{t2, t1, t0}</td>
</tr>
<tr>
<td>t4 ← - t3 t2</td>
<td>{t2, t1, t3}</td>
</tr>
<tr>
<td>t3 ← + t1 t3</td>
<td>{t1, t3}</td>
</tr>
<tr>
<td>t5 ← + t3 t3</td>
<td>{t3}</td>
</tr>
</tbody>
</table>

Liveness range of t1: \{3,4,5,6\}  Liveness range of t3: \{5,6,7\}
Register Allocation

If I have only 2 registers (r1 and r2), how do I allocate them?

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 ← int 2</td>
<td>{}</td>
</tr>
<tr>
<td>t1 ← + t0 t0</td>
<td>{t0}</td>
</tr>
<tr>
<td>t2 ← * t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>t1 ← + t1 t2</td>
<td>{t1, t2}</td>
</tr>
</tbody>
</table>
Register Allocation

If I have only 2 registers (r1 and r2), how do I allocate them?

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 ← int 2</td>
<td>{}</td>
</tr>
<tr>
<td>t1 ← + t0 t0</td>
<td>{t0}</td>
</tr>
<tr>
<td>t2 ← * t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>t1 ← + t1 t2</td>
<td>{t1, t2}</td>
</tr>
</tbody>
</table>
Register Interference Graph

- One node for each variable live in a section of code
- Edges connect nodes when they are live simultaneously

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 t0 ← int 1</td>
<td>{}</td>
</tr>
<tr>
<td>2 t1 ← int 2</td>
<td>{t0}</td>
</tr>
<tr>
<td>3 t2 ← + t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>4 t3 ← * t2 t0</td>
<td>{t2, t1, t0}</td>
</tr>
<tr>
<td>5 t4 ← - t3 t2</td>
<td>{t2, t1, t3}</td>
</tr>
<tr>
<td>6 t3 ← + t1 t3</td>
<td>{t1, t3, t4}</td>
</tr>
<tr>
<td>7 t5 ← + t3 t3</td>
<td>{t3, t4}</td>
</tr>
</tbody>
</table>

assume $LIVE_{out}$: {t4}
Register Interference Graph

- One node for each variable live in a section of code
- Edges connect nodes when they are live simultaneously

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 t0 ← int 1</td>
<td>{}</td>
</tr>
<tr>
<td>2 t1 ← int 2</td>
<td>{t0}</td>
</tr>
<tr>
<td>3 t2 ← + t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>4 t3 ← * t2 t0</td>
<td>{t2, t1, t0}</td>
</tr>
<tr>
<td>5 t4 ← - t3 t2</td>
<td>{t2, t1, t3}</td>
</tr>
<tr>
<td>6 t3 ← + t1 t3</td>
<td>{t1, t3, t4}</td>
</tr>
<tr>
<td>7 t5 ← + t3 t3</td>
<td>{t3, t4}</td>
</tr>
</tbody>
</table>

assume $LIVE_{out}$: {t4}
Register Allocation via ?

- Can we analyze the RIG to determine which variables get which registers?
Register Allocation via ?

- Can we analyze the RIG to determine which variables get which registers?
- This should remind you of a certain CS theory problem...
Register Allocation via ?

- Can we analyze the RIG to determine which variables get which registers?
- This should remind you of a certain CS theory problem...
  
- That’s right, it’s Graph coloring
- The number of colors ($k$) is your number of registers
Register Allocation via?

- Can we analyze the RIG to determine which variables get which registers?
- This should remind you of a certain CS theory problem...

- That’s right, it’s Graph coloring
- The number of colors ($k$) is your number of registers

- If the RIG is $k$-colorable, you can find registers for all variables in it
- NP-Complete solution... usually we use heuristics
Register Allocation via Graph Coloring

Find a coloring with $k = 3$ (allocate 3 registers...)
Spilling!
Spilling to Relieve Register Pressure

If the RIG is not $k$-colorable, we must find registers to spill.
Spilling to Relieve Register Pressure

- If the RIG is not \( k \)-colorable, we must find registers to spill
- Concept: Kill variables
  - Save a variable back to memory so it is no longer alive
  - You free up the register it was using so another can use it instead
Spilling to Relieve Register Pressure

- If the RIG is not \(k\)-colorable, we must find registers to spill

- **Concept** Kill variables
  - Save a variable back to memory so it is no longer alive
  - You free up the register it was using so another can use it instead

- This is essentially breaking up the liveness range!
Spilling to Relieve Register Pressure

- If the RIG is not $k$-colorable, we must find registers to *spill*

- **Concept** Kill variables
  - Save a variable back to memory so it is no longer alive
  - You free up the register it was using so another can use it instead

- This is essentially breaking up the liveness range!

- This means more calls to memory...
  - Common compilers can get held up here trying to be optimal
Spilling and Filling

1. Select a variable to spill
   - What is the ideal variable to spill?
   - **Heuristic**: The one with the most edges
2. Push it onto the stack
   - Don’t forget your symbol table
3. Rerun coloring algorithm, spill until you can $k$-color the RIG
4. When you need the variable again, bring it back from the stack
   *(fill a register)*
Spilling and filling in TAC

1. Filling requires a register...

2. Create a new temporary to store the variable
   - This means a different liveness range (it’s a new virtual register!)

3. You can rerun your liveness analysis and regenerate your RIG

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( t_0 \leftarrow \text{int 1} )</td>
<td>{}</td>
</tr>
<tr>
<td>2 ( t_1 \leftarrow \text{int 2} )</td>
<td>{t_0}</td>
</tr>
<tr>
<td>3 ( t_2 \leftarrow + t_1 t_0 )</td>
<td>{t_1, t_0}</td>
</tr>
<tr>
<td>4 ( t_3 \leftarrow * t_2 t_0 )</td>
<td>{t_2, t_1, t_0}</td>
</tr>
<tr>
<td>5 ( t_4 \leftarrow - t_3 t_2 )</td>
<td>{t_2, t_1, t_3}</td>
</tr>
<tr>
<td>6 ( t_3 \leftarrow + t_1 t_3 )</td>
<td>{t_1, t_3, t_4}</td>
</tr>
<tr>
<td>7 ( t_5 \leftarrow + t_3 t_3 )</td>
<td>{t_3, t_4}</td>
</tr>
</tbody>
</table>

assume \( \text{LIVE}_{\text{out}}: \) \{t_4\}

Compiler Construction
Spilling and filling in TAC

1. Filling requires a register...
2. Create a new temporary to store the variable
   - This means a different liveness range (it’s a new virtual register!)
3. You can rerun your liveness analysis and regenerate your RIG (spill t1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 t0 ← int 1</td>
<td>{}</td>
</tr>
<tr>
<td>2 t1 ← int 2</td>
<td>{t0}</td>
</tr>
<tr>
<td>3 t2 ← + t1 t0</td>
<td>{t1, t0}</td>
</tr>
<tr>
<td>3a t6 ← t1</td>
<td>{t2, t0}</td>
</tr>
<tr>
<td>4 t3 ← * t2 t0</td>
<td>{t2, t6, t0}</td>
</tr>
<tr>
<td>5 t4 ← - t3 t2</td>
<td>{t2, t6, t3}</td>
</tr>
<tr>
<td>6 t3 ← + t6 t3</td>
<td>{t6, t3, t4}</td>
</tr>
<tr>
<td>7 t5 ← + t3 t3</td>
<td>{t3, t4}</td>
</tr>
</tbody>
</table>

assume $LIVE_{out}$: {t4}
Remarks

- **Register Interference Graph** constructed from liveness
  - Indicates when variables are alive simultaneously
- **Note** The number of RIG nodes increases when you spill, but the edges are more sparse
Remarks

- **Register Interference Graph** constructed from liveness
  - Indicates when variables are alive simultaneously
- **Note** The number of RIG nodes increases when you spill, but the edges are more sparse
- **Cool ASM** places artificial importance on good allocation
  - Less pressure on x86-64, but references to memory are extremely expensive
Remarks

- **Register Interference Graph** constructed from liveness
  - Indicates when variables are alive simultaneously

- **Note** The number of RIG nodes increases when you spill, but the edges are more sparse

- Cool ASM places artificial importance on good allocation
  - Less pressure on x86-64, but references to memory are extremely expensive

- Some registers are *necessarily* pre-colored
Remarks

- **Register Interference Graph** constructed from liveness
  - Indicates when variables are alive simultaneously

- **Note** The number of RIG nodes increases when you spill, but the edges are more sparse

- Cool ASM places artificial importance on good allocation
  - Less pressure on x86-64, but references to memory are extremely expensive

- Some registers are *necessarily* pre-colored
  - e.g., x86-64 has rax as a return register
  - Some x86-64 instructions always use certain registers (rdtsc uses eax and edx)
Linking and Loading

- For PA5, your compiler’s output is a single assembly file

- Real compilers support multiple compilation units (e.g., many files) and must pull them all together in a process called linking
  - Linking involves resolving reference from one object or compilation unit to another

- We also want to support shared libraries through dynamic linking
Separate Compilation

- Compile different parts of your program at different times

- Then, **link** them together later

- This is a big win:
  - Faster compile times on small changes
  - Good software engineering practice (modularity)
  - Different parts can be independently developed (libraries)

- All major language and big projects use separate compilation
Compilation Units

- A compiled program fragment is called an **object file**

- An object file contains:
  - Code (for methods, expressions, etc.)
  - Variables (e.g., global variable locations/values)
  - Debugging information (symbolic information)
  - References to code and data that appear elsewhere (e.g., how to get `printf`)
  - Table that organize this information

- Object files are an implicit part of interpreters
Two Big Tasks

- The operating system uses virtual memory so every program starts at a standard (virtual) address (e.g., address 0)

- **Linking** involves two tasks
  - Linking the code and data from each object file to a particular fixed virtual address
  - Resolving references (e.g., to variable locations or jump target labels) so they point to concrete and correct virtual addresses
Relocatable Object Files

For this to work, a relocatable object file comes equipped with three tables

- **Import Table**: points to places in the code where an external *symbol* is referenced
  - List of pairs: (external_symbol, code_location)

- **Export Table**: points to symbol definitions in the code that are exported for use by others
  - List of pairs: (internal_symbol, code_location)

- **Relocation Table**: points to places in the code where local symbols are referenced
  - List of pairs: (internal_symbol, code_location)
Tables everywhere
C/ASM Example

```c
1   extern double sqrt (double x);
2   static double temp = 0.0;
3   double quadratic (double a, b, c) {
4       temp = b*b - 4.0*a*c;
5       if ( temp >= 0.0) { goto has_roots ; }
6       return Invalid_Argument;
7     has_roots:
8       return (-b + sqrt (temp)) / (2.0*a);
9 }```

Imports

```c
extern double sqrt (double x);
static double temp = 0.0;
double quadratic (double a, b, c) {
    temp = b*b - 4.0*a*c;
    if ( temp >= 0.0) { goto has_roots ; }
    return Invalid_Argument ;
    has_roots :
    return (-b + sqrt (temp)) / (2.0*a);
}
```

Import Table:
Replace address used at 0x1008 with final location of `sqrt`
Exports

1 extern double sqrt (double x);
2 static double temp = 0.0;
3 double quadratic (double a, b, c) {
4    temp = b*b - 4.0*a*c;
5    if ( temp >= 0.0) { goto has_roots ; }
6    return Invalid_Argument ;
7 has_roots:
8    return (-b + sqrt (temp)) / (2.0*a);
9 }

Export Table:
We provide quadratic. If others want it, find where 0x0200 is finally relocated to.
extern double sqrt (double x);
static double temp = 0.0;
double quadratic (double a, b, c) {
    temp = b*b - 4.0*a*c;
    if ( temp >= 0.0) { goto has_roots ; }
    return Invalid_Argument ;
has_roots :
    return (-b + sqrt (temp)) / (2.0*a);
}

Relocation Table:
Find location of temp. Replace address at 0x600 with location of temp
0x0600  r1 = ld loc temp
0x0604  jgz r1 loc has_roots
Summary

► Your relocatable object file: main.o
  ► Exports main(), imports sqrt(), relocations...

► Your math library: math.o
  ► Exports sqrt(), relocations
  ► Libraries can have imports: example?
  ► In Unix, math.o lives in libmath.a (use -lmath to find it with gcc)

► The linker reads them in, picks a fixed final relocation address for all code and data (first pass)

► then goes through and modifies every instruction with a symbol reference (second pass)
Big Link Example
Big Link Example (Answers)

Relocatable object files

A
- Imports
  - M
  - M
- Exports
  - X
- Relocation
- Code
  - r1 := &M
  - call M
- Data
  - X:

B
- Imports
  - X
- Exports
  - M
- Relocation
- Code
  - ...
  - r1 := &L (1000)
  - r2 := Y (400)
  - r3 := X
- Data
  - L:
  - M:
  - Y:

Executable object file

Code
- ...
- r1 := &M (2300)
- call M (2300)

- ...
- r1 := &L (1800)
- r2 := Y (3900)
- r3 := X (3300)

L:
M:

Data
- X:
- Y:
Dynamic Linking

Think about our math example: if every program links against `math.o` to use `sqrt()`, are there any drawbacks?

Hint: how large is the final `.exe` if it links `math.o` vs. not linking?

What if multiple programs use `math.o`?
Dynamic Linking

▸ Idea: shared libraries (.so) or dynamically linked libraries (.dll) use virtual memory so that multiple programs can share the same libraries in main memory
  ▸ Load the library into physical memory once
  ▸ Each program using it has a virtual address \( V \) that points to it
  ▸ During dynamic linking, resolve references to library symbols using that virtual address \( V \)

▸ What could go wrong?
Relocations in the DLL

- Since we are sharing the code to math.dll, we cannot set its relocations separately for each client.
- If math.dll contains a jump to math_label, that must be resolved to the same location (e.g., 0x1234) for all clients.
  - Recall: we only get to patch instructions once.
  - Every thread/program that uses the library shares the same patched code.

- 2 solutions:
  - Every program using math.dll agrees to put it at virtual address location \( X \) (bad).
  - or math.dll code is constructed to have no relocations in its code segment.
Position-independent Code

- Rather than using 0x1000: jump 0x1060, use jump PC+0x60
  - This code can be relocated to any address (i.e., any virtual address)
  - This is called position-independent code (PIC)