CS412/413

Introduction to Compilers
Radu Rugina

Lecture 32: Finishing Code Generation
14 Apr 03

Analysis and Optimizations

- Dataflow analysis reasons about variables and values
- Records (objects) consist of a collection of variables (fields) – analysis must separately keep track of individual fields
- Difficult analysis for heap-allocated objects
  - Object lifetime outlives procedure lifetime
  - Need to perform inter-procedural analysis
- Constructors/destructors: must take into account their effects

Class Hierarchy Analysis

- Method calls = dynamic, via dispatch vectors
  - Overhead of going through DV
  - Prohibits function inlining
  - Makes other inter-procedural analyses less precise
- Static analysis of dynamic method calls
  - Determine possible methods invoked at each call site
  - Need to determine principal types of objects at each program point (Class Hierarchy Analysis)
  - If analysis determines object o is always of type T (not subtype), then it precisely knows the code for o.foo()
- Optimizations: transform dynamic method calls into static calls, inline method calls

Putting Things Together

- Accessing variables
  - Global variables: using their static addresses
  - Function arguments and spilled variables (local variables and temporaries): using frame pointer
  - Variables assigned to registers: using their registers
- Instruction selection
  - Need to know which variables are in registers and which variables are spilled on stack
- Register allocation
  - No need to allocate a register to a value inside a tile

Code Generation Flow

- Start with low-level IR code
- Build DAG of the computation
  - Access global variables using static addresses
  - Access function arguments using frame pointer
  - Assume all local variables and temporaries are in registers (assume unbounded number of registers)
- Generate abstract assembly code
  - Perform tiling of DAG
- Register allocation
  - Live variable analysis over abstract assembly code
  - Assign registers and generate assembly code

Example

Program

```
array[int] a

function f:(int x) {
  int i;
  ...
  a[x+i] = a[x+i] + 1;
  ...
}
```

Low IR

```
t1 = x+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t3 = x+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

Accesses to Function Arguments

```
t1 = x+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t3 = x+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

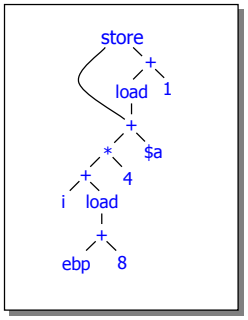
⇒

```
t4 = ebp+8
t5 = *t4
t1 = t5+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t6=ebp+8
t7 = *t6
t3 = t7+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```


DAG Construction

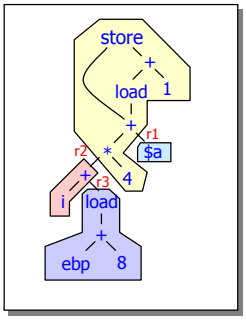
```
t4 = ebp+8
t5 = *t4
t1 = t5+i
t1 = t1*4
t1 = a+t1
t2 = *t1
t2 = t2+1
t6=ebp+8
t7 = *t6
t3 = t7+i
t3 = t3*4
t3 = a+t3
*t3 = t2
```

⇒

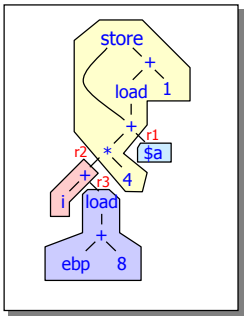


Tiling

- Find tiles
  - Maximal Munch
  - Dynamic programming
- Temporaries to transfer values between tiles
- No temporaries inside any of the tiles



Abstract Assembly Generation



⇒ Abstract Assembly

```
mov $a, r1
mov 8(%ebp), r3
mov i, r2
add r3, r2
add $1, (r1,r2,4)
```


Register Allocation

Abstract Assembly

```
mov $a, r1
mov 8(%ebp), r3
mov i, r2
add r3, r2
add $1, (r1,r2,4)
```

⇒ Live Variables

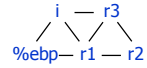
```
                    {%ebp, i}
mov $a, r1
                    {%ebp,r1,i}
mov 8(%ebp), r3
                    {r1, r3, i}
mov i, r2
                    {r1,r2,r3}
add r3, r2
                    {r1,r2}
add $1, (r1,r2,4)
                    {}
```


Register Allocation

Live Variables

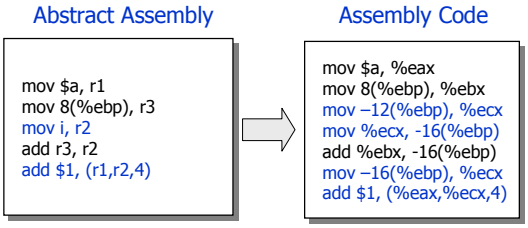
```
                    {%ebp, i}
mov $a, r1
                    {%ebp,r1,i}
mov 8(%ebp), r3
                    {r1, r3, i}
mov i, r2
                    {r1,r2,r3}
add r3, r2
                    {r1,r2}
add $1, (r1,r2,4)
                    {}
```

- Build interference graph



- Allocate registers:
  eax: r1, ebx: r3
  i, r2 spilled to memory

Assembly Code Generation



Register allocation results:
eax: r1; ebx: r3; i, r2 spilled to memory

Other Issues

- Translation of function calls / functions
  - Pre-call/post-call code
  - Prologue/epilogue code
- Saved registers
  - If caller-save register is live after call, must save it before call and restore it after call
  - If callee-save register is allocated within a procedure, must save it at procedure entry and restore at exit
- Objects
  - Dispatch vectors (static)
  - Indirect method calls, implicit object parameter
  - Accessing object fields

Advanced Code Generation

- Modern architectures have complex features
- Compiler must take them into account to generate good code
- Features:
  - Pipeline: several stages for each instruction
  - Superscalar: multiple execution units execute instructions in parallel
  - VLIW (very long instruction word): multiple execution units, machine instruction consists of a set of instructions for each unit

Pipeline

- Example pipeline:
  - Fetch
  - Decode
  - Execute
  - Memory access
  - Write back

- Simultaneously execute stages of different instructions

Stall the Pipeline

- It is not always possible to pipeline instructions
- Example 1: branch instructions

- Example 2: load instructions

Filling Delay Slots

- Some machines have delay slots
- Compiler can generate code to fill these slots and keep the pipeline busy
- Branch instructions
  - Fill delay slot with instruction which dominates the branch, or which is dominated by the branch
  - Compiler must determine that it is safe to do so
- Load instructions
  - If next instruction uses result, it will get the old value
  - Compiler must re-arrange instructions and ensure next instruction doesn't depend on results of load

Superscalar

- Processor has multiple execution units and can execute multiple instruction simultaneously
- … only if it is safe to do so!
- Hardware checks dependencies between instructions
- Compiler can help: generate code where consecutive instructions can execute in parallel
  - Again, need to reorder instructions

VLIW

- Machine has multiple execution units
- Long instruction: contains instructions for each execution unit
- Compiler must parallelize code: generate a machine instruction which contains independent instructions for all the units
- If cannot find enough independent instructions, some units will not be utilized
- Compiler job very similar to the transformation for superscalar machines

Instruction Scheduling

- Instruction scheduling = reorder instructions to improve the parallel execution of instructions
  - Pipeline, superscalar, VLIW
- Essentially, compiler detects parallelism in the code
- Instruction Level Parallelism (ILP) = parallelism between individual instructions
  - Instruction scheduling: reorder instructions to expose ILP

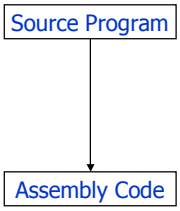
Instruction Scheduling

- Many techniques for instruction scheduling
- List scheduling
  - Build dependence graph
  - Schedule an instruction if all its predecessors have been scheduled
  - Many choices at each step: need heuristics
- Scheduling across basic blocks
  - Move instructions past control flow split/join points
  - Move instruction to successor blocks
  - Move instructions to predecessor blocks

Instruction Scheduling

- Another approach: try to increase basic blocks
  - Then schedule the large blocks
- Trace scheduling
  - Use profiling to find common execution paths
  - Combine basic blocks in the trace into a larger block
  - Schedule the trace
  - Problem: need cleanup code if program leaves trace
- Duplicate basic blocks
- Loop unrolling

Instruction Scheduling

- Can also schedule across different iterations of loops
- Software pipelining
  - Overlap loop iterations to fill delay slots
  - If latency between instructions i1 and i2 in some loop iteration, change loop so that i2 uses results of i1 from previous iteration
  - Need to generate additional code before and after the loop

Where We Are