CS412/413

Introduction to Compilers

Radu Rugina

Lecture 29: Finishing Code Generation

08 Apr 02

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 = addr a
t2 = x+i
t2 = t2*4
t1 = t1+t2
t3 = [t1]
t3 = t3+1
t4 = addr a
t5 = x+i
t5 = t5*4
t4 = t4+t5
[t4] = t3
```

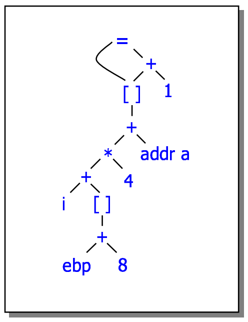
Accesses to Function Arguments

```
t1 = addr a
t2 = x+i
t2 = t2*4
t1 = t1+t2
t3 = [t1]
t3 = t3+1
t4 = addr a
t5 = x+i
t5 = t5*4
t4 = t4+t5
[t4] = t3
```

```
t1 = addr a
t6 = ebp+8
t7 = [t6]
t2 = t7+i
t2 = t2*4
t1 = t1+t2
t3 = [t1]
t3 = t3+1
t4 = addr a
t8=ebp+8
t9 = [t8]
t5 = t9+i
t5 = t5*4
t4 = t4+t5
[t4] = t3
```

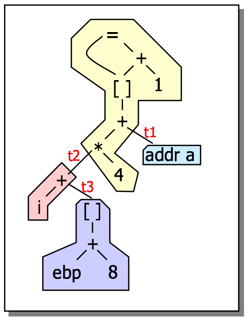
DAG Construction

```
t1 = addr a
t6 = ebp+8
t7 = [t6]
t2 = t7+i
t2 = t2*4
t1 = t1+t2
t3 = [t1]
t3 = t3+1
t4 = addr a
t8=ebp+8
t9 = [t8]
t5 = t9+i
t5 = t5*4
t4 = t4+t5
[t4] = t3
```

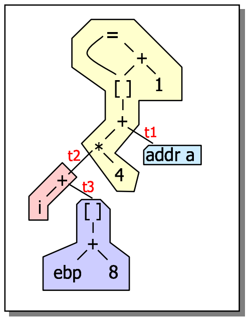


Tiling

- 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 addr a, t1
mov 8(%ebp), t3
mov i, t2
add t3, t2
add 1, (t1,t2,4)
```

Register Allocation

Abstract Assembly

```
mov addr a, t1
mov 8(%ebp), t3
mov i, t2
add t3, t2
add 1, (t1,t2,4)
```

Live Variables

```
                {%ebp, i}
mov addr a, t1
                {%ebp,t1,i}
mov 8(%ebp), t3
                {t1, t3, i}
mov i, t2
                {t1,t2,t3}
add t3, t2
                {t1,t2}
add 1, (t1,t2,4)
                {}
```

Register Allocation

Live Variables

```
                {%ebp, i}
mov addr a, t1
                {%ebp,i,t1}
mov 8(%ebp), t3
                {t1, t3, i}
mov i, t2
                {t1,t2,t3}
add t3, t2
                {t1,t2}
add 1, (t1,t2,4)
                {}
```

- Build interference graph

$$\begin{tikzcd} & i \arrow[r, dash] \arrow[dl, dash] \arrow[d, dash] & t3 \arrow[dl, dash] \arrow[d, dash] \\ \%ebp \arrow[r, dash] & t1 \arrow[r, dash] & t2 \end{tikzcd}$$

- Allocate registers:
  eax: t1, ebx: t3
  i, t2 spilled to memory

Assembly Code Generation

Abstract Assembly

```
mov addr a, t1
mov 8(%ebp), t3
mov i, t2
add t3, t2
add 1, (t1,t2,4)
```

Assembly Code

```
mov addr a, %eax
mov 8(%ebp), %ebx
mov –12(%ebp), %ecx
mov %ecx, -16(%ebp)
add %ebx, -16(%ebp)
mov –16(%ebp), %ecx
add $1, (%eax,%ecx,4)
```

Register allocation results:
eax: t1; ebx: t3; i, t2 spilled to memory

Other Issues

- Translation of function calls
  - Pre-call code
  - Post-call code
- Translation of functions
  - Prologue code
  - 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

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

Fetch	Dec	Exe	Mem	WB

- Simultaneously execute stages of different instructions

Instr 1	Fetch	Dec	Exe	Mem	WB		
Instr 2		Fetch	Dec	Exe	Mem	WB	
Instr 3			Fetch	Dec	Exe	Mem	WB

Stall the Pipeline

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

Branch	Fetch	Dec	Exe	Mem	WB			
Target				Fetch	Dec	Exe	Mem	WB

- Example 2: load instructions

Load	Fetch	Dec	Exe	Mem	WB		
Use			Fetch	Dec	Exe	Mem	WB

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

$$\begin{tikzcd}\boxed{\text{Source Program}} \arrow[d, "\text{ü}"] \\ \boxed{\text{Assembly Code}}\end{tikzcd}$$