SBCL Peephole Optimization

CMPS 112
Scott Chatham

Peephole Optimization

Peephole optimization is an optimization technique that looks at sliding groups of instructions, the peephole, and performs various optimization techniques. These optimizations can be of several types such as:

1. redundant instruction elimination (unreachable code/ duplicate instructions)
2. flow of control optimization (jumps to jumps)
3. algebraic simplifications (multiplication instead of of exponentiation, x*x instead of pow(x,2))
4. use of machine specific idioms (special instructions)

Issues - SBCL is large

- 90,000 lines of lisp code implementing the ‘standard library’, excluding the Common Lisp Object System (CLOS)
- 60,000 lines of lisp code implementing the compiler (and related subsystems, such as the debugger internals)
- between 10,000 and 20,000 lines of lisp code per architecture backend implementing the code generators and low-level assembly routines
- 20,000 lines of lisp code implementing CLOS
- 20,000 lines of lisp code implementing contributed modules or ‘extras’
- 35,000 lines of C and assembly code, for services such as signal handling and garbage collection
- 30,000 lines of shell and lisp code for regression tests

~ 270k LOC

Issues - rusty CL and assembly

- it's been a while since I've used CL beyond tinkering, not quite a year
- even longer for assembly

- CL has great documentation and looking up forgotten stuff was quick and painless
- assembly not so much, initially limit to easy optimizations such as redundant MOV ops:

```
MOV EAX, EDX     or     MOV EAX, 2
MOV EDX, EAX            CMP EDX, EAX
```

Test Case

- (defun square (x) (* x x))
- (compile-file “~/test.lisp” :trace-file t)

  generates a trace file containing all intermediate representations as well as the final assembly generated

```
;       27:         8BDA            MOV EBX, EDX
;       29:         895DFC          MOV [EBP-4], EBX
;       2C:         8BD3            MOV EDX, EBX
;       2E:         8BFB            MOV EDI, EBX
;       30:         E800000000      CALL L0
;       35: L0:     8B5DFC          MOV EBX, [EBP-4]
;       38:         8BE5            MOV ESP, EBP
;       3A:         F8              CLC
;       3B:         5D              POP EBP
;       3C:         C3              RET
```

Basic Approach

- the idea:
    1. buffer up instructions
    2. run optimization pass
    3. output the instructions
- pretty straightforward right?
- remember those 270k LOC?
- what do I buffer and where do I do it?

IR2

- SBCL has two intermediate representations: ICR (Implicit Continuation Representation), and VMR (Virtual Machine Representation)
- VMR is the one we care about, it's the final form of the program before target code is generated
- it has two major constructs:
  1. VOPs (Virtual OPerations) - high level assembly encoded in CL
  2. TNs (Temporary Names) - represent eventual storage locations such as registers

Implementation #1

- hijacked the trace routine described earlier which is located in codegen.lisp
- the peephole pass was put in assem.lisp where the DS of the trace routine was located
- easily allowed buffering of VOPs and optimizing them
- problem: no way to emit the results
- dead end, but all of the peephole pass and MOV optimization code is reusable

Implementation #2

- moved the buffering and optimization pass into the code generation area, codegen.lisp
- buffered up VOPs as code generation processed basic blocks (independent units of code)
- ran optimization pass on buffer as before, but afterwards the optimized sequence is emitted as assembly
- problem: not all VOPs are buffered at the same time, only VOPs within the basic block being processed at that moment

```
;       27:       8955FC          MOV [EBP-4], EDX
;       2A:       8BD3            MOV EDX, EBX
;       2C:       8BFB            MOV EDI, EBX
;       2E:       E800000000      CALL L0
;       33: L0:   8B5DFC          MOV EBX, [EBP-4]
;       36:       8BE5            MOV ESP, EBP
;       38:       F8              CLC
;       39:       5D              POP EBP
;       3A:       C3              RET
```

Implementation #2.5

- solution: lift VOP emission out of the basic block loop, success!

```
;       27:         8BDA            MOV EBX, EDX
;       29:         895DFC          MOV [EBP-4], EBX
;       2C:         8BD3            MOV EDX, EBX
;       2E:         8BFB            MOV EDI, EBX
;       30:         E800000000      CALL L0
;       35: L0:     8B5DFC          MOV EBX, [EBP-4]
;       38:         8BE5            MOV ESP, EBP
;       3A:         F8              CLC
;       3B:         5D              POP EBP
;       3C:         C3              RET
```

```
;       27:         8955FC          MOV [EBP-4], EDX
;       2A:         8BFA            MOV EDI, EDX
;       2C:         E800000000      CALL L0
;       31: L0:     8B5DFC          MOV EBX, [EBP-4]
;       34:         8BE5            MOV ESP, EBP
;       36:         F8              CLC
;       37:         5D              POP EBP
;       38:         C3              RET
```

Conclusion

- fun project, interesting subject
- lots of work left to do:

1. bug fixes - works on simple case, not so much on larger ones
2. pattern matching language
3. correctness of optimization

Questions?