SBCL Peephole Optimization

Scott Chatham, schatham@ucsc.edu

March 12th, 2014

Introduction

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)

My goal with the project was to implement such a peephole optimization pass for the Steel Bank Common Lisp compiler. The main optimization I was interested in getting working was eliminating redundant move operations as they are fairly prevalent in SBCL generated code. Code such as the following could be eliminated depending on the surrounding context:

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

The peephole pass would recognize the redundant MOVs and then search through the next few instructions to ensure no instructions relied on them or if they did, possibly alter the instructions to account for the eliminated MOV.

Issues

A big issue was how large SBCL is. According to Rhodes[1], as of 2008, SBCL looked like this:

- 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.

This is probably pretty small for a production level compiler, but it's still orders of magnitudes larger than any other project I have previously worked on. Also, besides a few old papers describing the overall structure of the compiler, all I had to go on was the code and surrounding documentation itself. So knowing where to look and understanding the system was a large hurdle. Luckily, the papers did point me in the right direction and narrowed the scope down to a few thousand lines of code, but this is still quite a bit larger than most code I've worked with and it's all foreign.

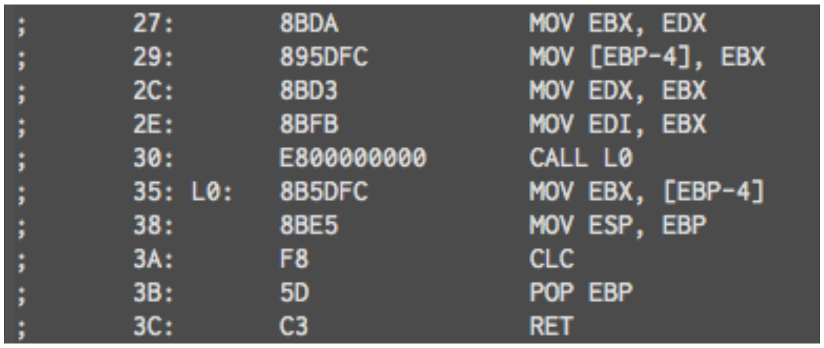
Another issue was it had been a while since I had really worked with CL and assembly. The CL part wasn't so bad as I just needed to look up bits I had forgotten. The difficult part was grokking all the macro usage, which I still haven't, as it got pretty advanced. On the assembly side, several years have passed since I first learned assembly and I haven't used it since. This is one of the reasons I aimed for eliminating redundant MOV instructions as it seemed a fairly simple case. I also mostly restricted development to a virtual machine running x86 linux so I didn't have to worry about usage of different sized registers such as RDX (64 bit) and EDX (32 bit).

Finally, development was made difficult as certain modifications would break the entire system and it needed to be restarted. This was quick, but still a pain at times. Also, some changes required recompiling the entire compiler multiple times as errors halted compilation and there was no debugging available because the system wasn't built yet. This only happened occasionally and in most cases it wasn't an issue thanks to CL's incremental development. If the changes were local they could just be hot patched into the running system which was a huge boon and time saver.

Approach

The first thing I did was find all documentation on the compiler that I could and start getting the layout of the land. Once I found the general area of where I thought the changes would be I started reading through the code to build an image of the process it went through. I eventually got an idea of how I could start implementing, but first I needed a test case.

SBCL provides an option to output all the intermediate representations of the compiler as well as the assembly code generated during compilation. Using this, I compiled a simple square function, (defun square (x) (* x x)), whose disassembly looked like this:



```
;       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
```

I omitted some error checking code before and after these instructions, but if you look at offsets 27 and 29 you can see a MOV followed by another MOV to memory. Here's a case where we can eliminate the first MOV and change the argument of the second. We'll have to fix the instructions following as they relied on the first MOV being performed, but that should be doable. Now we can talk about what we need to do.

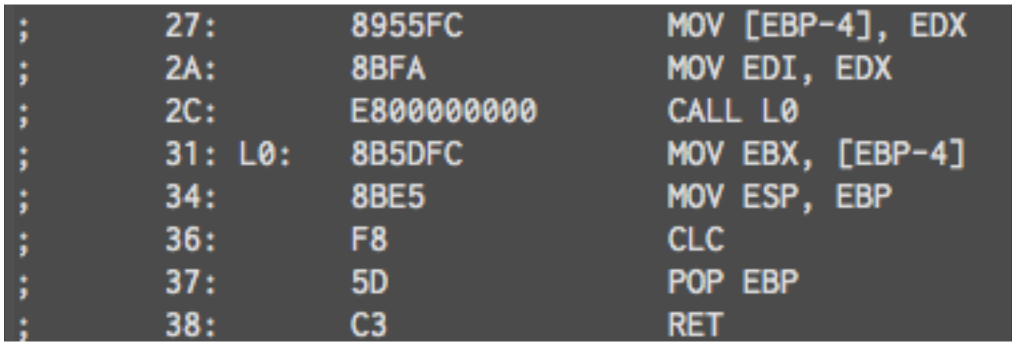
The main idea is to buffer up all the instructions so we have them in one location, perform the optimization pass until no more optimizations are found or a certain amount of time has gone by, and finally emit the optimized code. The difficult part was determining where the buffering would take place and what to buffer. I ended up trying a few different approaches that I'll discuss next. Both approaches worked on the highest level of instructions within the compiler: VOPs (Virtual OPerations), and TNs (Temporary Names). The VOPs represent the instructions and the TNs represent eventual storage locations such as registers. These two constructs are part of the second form of SBCL's intermediate representation, known as VMR (Virtual Machine Representation), and this is the final form of the program before the target code is generated.

Results

The first approach I tried hijacked the previously discussed compilation trace routine to intercept all the VOPs and enable buffering. This approach was simple as all the information needed is contained in one place within the trace routine. All I had to do was add a buffer to the segment, the data structure that holds the current state of the assembler, and add the VOPs as the trace saw them. I implemented a basic peephole pass and MOV optimization that successfully recognized the redundant MOV ops and fixed them. I printed out the VOPs before and after the optimization to test if it was working. This worked to an extent as it allowed me to examine the instructions all at once and manipulate them. The problem was there was no way to go back and emit this optimized sequence of VOPs.

My second approach looked within the code generation function itself where the VOP emitters are called and tried to optimize them before that happened. I was able to buffer the VOPs and pass them to the optimizer before they were passed on and emitted. This also worked to an extent as I was now able to modify the instructions being output. The problem was that not all of the VOPs are available at this point in time. The code generation function loops over basic blocks, independent sections of code, and emits all of the instructions for each block independent of any other block. I could indeed make changes to the instructions, but I couldn't propagate the changes across the boundaries of the blocks. So in the disassembly listed for the square function, I couldn't modify the arguments of the two subsequent MOV ops because they passed through the optimizer at a later time.

I thought I was pretty close to something working at this point, I just needed to get all of the VOPs together. I was able to lift the VOP emission out of it's current place and replace it with a buffering of the VOPs. Once all the blocks have been processed and the VOPs have been buffered, we can pass them to the optimizer. Afterwards, we can call the VOP generators to emit the code as before. Lifting this process out of its previous location doesn't seem to have had any negative repercussions as no errors arise with the optimizer turned off and all tests pass. This approach worked and the disassembly of the optimized output looks like this:



```
;       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
```

And indeed, loading the compiled test file and executing a few square function calls produces the desired output. The optimizations currently only work in small test cases as it needs to be made a bit smarter in certain cases, but it's proof that it does work.

Conclusion

There seems to be two main approaches to implementing the peephole optimizer within SBCL: working at the VOP level as I did, and working at the assembly level once the instructions have been emitted, but before they're encoded in machine code. I'd like to explore the differences between these two approaches and see what kind of limitations there are. I'm sure there are tradeoffs to each and it would be interesting to explore them.

This was a really interesting project and I definitely plan to continue working on it. It didn't seem like my approach was going to pan out, but it came together in the end. I intended to implement more than one optimization, but this wasn't really feasible as it took all my time just to get the one working. Now that I have the basic structure in place I can explore other cases to implement and get an idea of how the system would work with multiple optimizations to perform.

There's also the eventual goal of writing a pattern matching language to actually implement the optimizations. The overall structures of the peephole pass and mov-optimization would need to be changed, but there are definitely ideas in each that could be taken out and generalized into useful constructs.

References

[1] Rhodes, Christophe. SBCL: A Sanely-Bootstrappable Common Lisp. http://www.doc.gold.ac.uk/~mas01cr/papers/s32008/sbcl.pdf