Yulun Zeng

When I first started, I was planning to do multiple iterations of dead code elimination, constant propagation, and constant folding. When I got that working, I wanted to try local value numbering.

When I do local value numbering, I spent a lot of time deciding which data structure to use. I did not hash the instruction at first, rather I focused on copy propagation and constant folding. But at some point, I was losing high level instructions even when I comment out the existing optimization. The code also became so buggy that I could proceed no further.

Then I decided to get rid of local value numbering, rather try register allocation only. I used a map to keep track of which register was used by which operand, but it didn't work well with memory references. When I encounter memory references, I started getting errors like
unsupported instruction `mov'
and
incorrect register `%r13' used with `l' suffix
I believe this problem as a result of poor communication between lowlevel codegen and register allocation, but at this time I really don't have any late hours left to fix anything.

I decided to go with my one and only working constant propagation + constant folding.
I use a map that keeps track of the constant value of registers and their constant values. But before that, there are several conditions to check:
There are 2 types of instructions, the ones with a destination and the ones without.
If we further divide theses two groups, there are 2 operands instructions like add, sub... and 1 operand instructions like move, localaddr...
And when we go further, the destination operand can be either memory reference, a plain virtual register with base value.
We should definitely not propagate constants to memory references and registers ranging from vr0 to vr9, but we can replaces other virtual registers with constants if such mapping exists.
Now we know which types of virtual registers can be replaced, we need to know when can we replace.
We must check the liveness of registers, if they are alive after a basic block, we don't want to replace them, because they might have been assigned a value that will be used later, such as a loop variable. If we replace those, the loop value will be stuck.
Next is updating or deleting the mapping. After each assignment, for example mov vr10, $3, the mapping of vr10 to constant will be updated. But when it's a mov vr10 vr11, we will delete the mapping for the sake of constant propagation(In my failed local value numbering code, I tried to keep a map<int, Operand>, but it didn't work well for memory references).
If we arrive at an add/sub/div/mul operation with 2 constant operands, we can perform a calculation, update the destination mapping, and ignore the instruction.
The next thing is do is repeat the constant propagation and constant folding for 10 times, this method effectively reduced roughly 12% lowlevel code for example 29 and 31, and the execution time on average shortened from ~1.8s to ~1.7s. It could be helpful for programs with a lot of constant calculations, but not so much for dynamic values.

Some snippets of before and after for the high level codes:
Before:

```
mov_l    vr16, $500
mov_l    vr10, vr16
localaddr vr16, $0
```

```
        mov_l    vr17, $0
        sconv_lq vr18, vr17
        mul_q    vr18, vr18, $4000
        add_q    vr19, vr16, vr18
        mov_l    vr20, $0
        sconv_lq vr21, vr20
        mul_q    vr21, vr21, $8
        add_q    vr22, vr19, vr21
        mov_q    vr1, vr22
After:
        mov_l    vr10, $500
        localaddr vr16, $0
        add_q    vr19, vr16, $0
        add_q    vr22, vr19, $0
        mov_q    vr1, vr22
---------------------------------------
Before:
        mov_l    vr13, $1
        mov_l    vr10, vr13
        mov_l    vr13, $11
        mov_l    vr11, vr13
        mov_l    vr13, $0
        mov_l    vr12, vr13
After:
        mov_l    vr10, $1
        mov_l    vr11, $11
        mov_l    vr12, $0
---------------------------------------
Before:
        mov_l    vr13, $2
        mov_l    vr10, vr13
        mov_l    vr13, $3
        mov_l    vr11, vr13
        mov_l    vr13, $0
        mov_l    vr12, vr13
        mov_l    vr13, $1
        add_l    vr14, vr10, vr13
        cmpeq_l  vr15, vr14, vr11
        cjmp_f   vr15, .L1
        mov_l    vr16, $1
        mov_l    vr12, vr16
After:
        cmpeq_l  vr15, $3, $3
        cjmp_f   vr15, .L1
        mov_l    vr12, $1
```

For programs that involve a lot of variable references like example29, there aren't that many constants to pass around, so the performance tend to be very poor for that. For programs like those, especially with so many loops and redundant calculations, using LVN to remove recomputation will play a big role. But unfortunately I could not get mine working.

I am aware that the efficiency will not have a significant increase until local register allocation is implemented, because accessing register is way faster than access the cache or memory. I gave my best.

Previous versions, work history: https://github.com/YunZng/compiler_code_gen