CS 1622: Code Generation & Register Allocation

Jonathan Misurda
jmisurda@cs.pitt.edu

Why is Code Generation Hard?
If the goal is to simply generate target code, we have already done this when we generated IR:
- Walk the AST and emit target code.
However, if we want to generate good target code, there are many things to consider.
Ultimately, the back end of the compiler is the repository of machine-specific knowledge.

We need to be able to assess among the multiple possible ways to encode a calculation, which one is best.

Arrays
Consider converting the following to machine code and data:
int A[low ... high];
A[i]++;

To deal with this array, we need to know the following things:
- width — width (size) of each element
- base — address of the first element
- low/high — lower/upper bound of subscript

Array Element Address
The address of element A[i] is then:
base + (i - low) \times width =
i \times width + (base-low\times width) =
i \times width + C1
Where C1 is a constant for this array.

Multidimensional Arrays
How should we store the data for a 2-Dimensional array?
Memory is one dimensional and so we must linearize our multi-dimensional arrays.

Two choices for how to do this:
- Row Major Order
- Column Major Order
Row Major Order

— Store data elements row by row

Blue elements are stored before \( A[i_1, i_2] \)

Address of Element \( A[i_1, i_2] \):

\[
= \text{base} + ((i_1 - \text{low}_1) \times N_2 + (i_2 - \text{low}_2)) \times \text{width}
\]

\[
= (i_1 \times N_2 + i_2) \times \text{width} + C_2R
\]

Column Major Order

— Store data elements row by row

Blue elements are stored before \( A[i_1, i_2] \)

Address of Element \( A[i_1, i_2] \):

\[
= \text{base} + ((i_2 - \text{low}_2) \times N_1 + (i_1 - \text{low}_1)) \times \text{width}
\]

\[
= (i_2 \times N_1 + i_1) \times \text{width} + C_2C
\]

Higher Dimensional Arrays

Row major: addressing a k-dimension array item (\( \text{low}_i = \text{base} = 0 \))

\[
A_k = A_{k-1} \times N_k + i_k \times \text{width}
\]

Column major: addressing a k-dimension array item (\( \text{low}_i = \text{base} = 0 \))

\[
A_k = i_k \times N_{k-1} \times N_{k-2} \times \ldots \times N_1 \times \text{width} + A_{k-1}
\]

C Arrays

C uses row major order:

```c
int fun(int p[][100])
{
    int a[100][100];
    ...
    a[i1][i2] = p[i1][i2] + 1;
    ...
}
```

Why is \( p[][100] \) allowed?
- The information is enough to compute \( p[i1][i2] \)'s address
  - \( A_k = (i_1 \times N_2 + i_2) \times \text{width} \) ...

Why is \( a[][100] \) not allowed?
- Need to allocate space

Java Arrays

Java doesn’t have 2+ dimensional arrays.

Arrays are arrays of arrays.

```java
int [][] a = new int[5][5];
```

Why Does it Matter?

Caching
Memory Hierarchy

SRAM: 1-25 ns (2-100 cycles)
DRAM: 60-120 ns
Magnetic disk: 10-20 million ns

Locality

How do we know what to include in the levels that are faster but smaller?
Use principles of locality:

- **Temporal locality**: What you use now, you will likely use again soon.
- **Spatial locality**: When you access an address, you will likely access its neighbors soon.

Caches Exploit Locality

For **temporal locality**, keep more recently used items closer to the processor. Less recently used items can be kept farther away.

For **spatial locality**, get items nearby referenced item at the same time as the requested item. (That is, don’t just bring what was requested but rather move larger blocks of contiguous memory.)

Cache Basics

Is address 16 in the cache?

If yes, we have a cache “hit”. If no, we have a cache “miss”.

Address 16 isn’t in the cache, so we must go to a farther away level of the memory hierarchy.

On a miss of our cache, we must go to main memory.
Cache Basics

Is address 16 in the cache?

If yes, we have a cache “hit”.
If no, we have a cache “miss”.

Address 16 isn’t in the cache, so we must go to a farther away level of the memory hierarchy.

On a miss of our cache, we must go to main memory.

Data can then be transferred between levels.

Row by Row vs. Col by Col

#define ROWS 20000
#define COLS 20000

int a[COLS][ROWS];
int main() {
    int i; int j;
    long long sum =0;
    for(i=0;i<COLS;i++)
        for(j=0; j<ROWS; j++)
            a[i][j]=rand()%10+1;
    for(i=0;i<ROWS;i++)
        for(j=0; j<COLS; j++)
            sum += a[i][j];
    return 0;
}

#define ROWS 20000
#define COLS 20000

int a[COLS][ROWS];
int main() {
    int i; int j;
    long long sum =0;
    for(i=0;i<ROWS;i++)
        for(j=0; j<COLS; j++)
            a[j][i]=rand()%10+1;
    for(i=0;i<ROWS;i++)
        for(j=0; j<COLS; j++)
            sum += a[j][i];
    return 0;
}

Results

gcc -m32 -o row roworder.c  
gcc -m32 -o col colorder.c

time ./row

real    0m15.979s  
user    0m14.651s  
sys     0m1.326s

time ./col

real    0m38.640s  
user    0m37.417s  
sys     0m1.212s

2.55x slower just by interchanging the loops!

Processing Boolean Expressions

Representation of True and False:
Like C:
0 – False
Anything Else – True
Alternative:
0 – False
-1 – True (-1 in Two’s complement is the string of all 1s)

Short Circuiting
E = (a < b) or (c < d and e < f)
if (a<b) goto TRUE_CODE
L1: if (c<d) goto L2
goto FALSE_CODE
L2: if (e<f) goto TRUE_CODE
goto FALSE_CODE

Processing Control Flow

Whenever we have forward control flow jumps (to locations we haven’t translated yet) we are unable to generate the target labels for the code to jump to.

There are two options:
• Do it in a single pass and resolve unknown jumps using backpatching
• Generate the code in one pass and then the labels in a second pass
Backpatching

Create a worklist of "holes" to fill in as we gain the information necessary to do so.

100: if (a < b) goto ___  Process this branch and add (100) to our worklist
101: a := a + 1
102: b := b + a
103: goto ___  Process this jump and add (103) to our worklist

100: if (a < b) goto 104  Process this branch and add (100) to our worklist
101: a := a + 1
102: b := b + a
103: goto ___  Process this jump and add (103) to our worklist
104: This is the first statement of the basic block (100) branches to. Go back and fill in the jump to 104.