CMPSCI 630 Systems **Fall 2019**

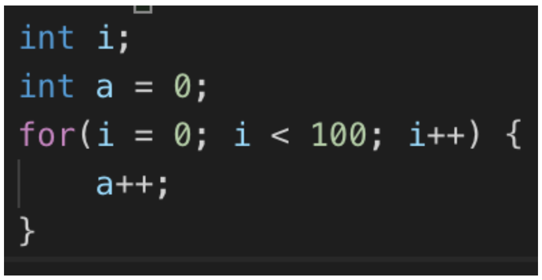
Lecture 4

Lecturer: Emery Berger *Scribe: Bryce Bodley-Gomes, Bochen Xu*

4.1 Automatic-Parallelization

We entered with the question of *Why is FORTRAN still in use?* The main reason is that it is fast, even when compared to other modern high level languages, but how? It turns out that because of the simple design of the language, it has been optimized for many years and because of the allowed pointer usage in the language (blocking aliasing in arrays), it is possible to determine dependencies.

Consider a for loop, it will be great if the program can make it parallel automatically, without the user writing all the concurrency code by oneself. But not all for loops can be parallelized while maintaining the same semantics. For example:



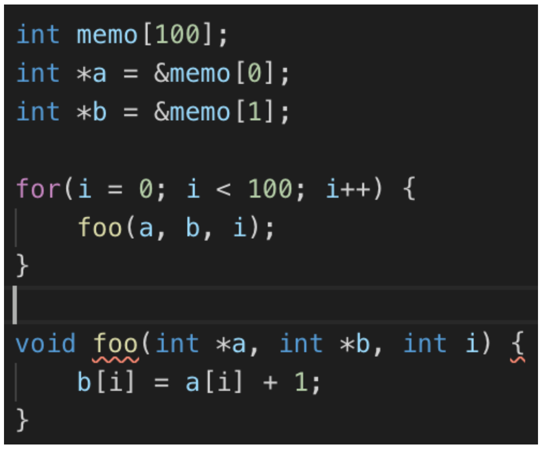
```c
int i;
int a = 0;
for(i = 0; i < 100; i++) {
    a++;
}
```

Figure 4.1: C code snippet

Theoretically a could be any integer value in between 1 and 100 after the code in Figure 4.1 runs. We refer to the race condition that happens between each iteration with loop-carried dependencies.

Sometimes the dependency is not obvious as in Figure 4.2. At first glance one may assume a and b are different independent entities, while in fact they are pointing to the same piece of memory. This is called aliasing, and for automatic parallelization to work, aliasing analysis, or equivalently, pointer analysis, is required to be able to ensure that there is no dependents for computations run in parallel (which it can't assure in c).

This brought us to the concept of parallel directives. These tools (Open MP Pragmas) allows linear code writing to be interpreted with parallelization without the programmer having to worry about concurrency logic. C/C++ kind of allows this sequential parallelization, but because of aliasing being allowed in the language, the correctness of results not guaranteed. FORTRAN doesn't allow aliasing of arrays, therefore it doesn't have this issue of potential dependencies, as a result it can guarantee that the code will execute correctly if loops for example are parallelized.



```c
int memo[100];
int *a = &memo[0];
int *b = &memo[1];

for(i = 0; i < 100; i++) {
    foo(a, b, i);
}

void foo(int *a, int *b, int i) {
    b[i] = a[i] + 1;
}
```

Figure 4.2: C code snippet

4.2 Static Analysis

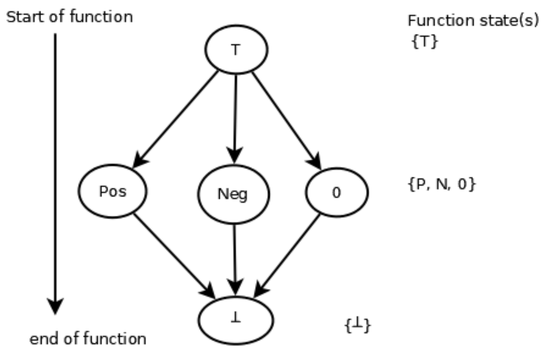


Figure 4.3: lattice of potential states for variable x

Trying to statically analyze the code execution of program to determine potential states has multiple possible approaches:
1. Try all potential inputs to the function (possibly infinite amount of inputs to analyse)
2. Try to map the states to a more limited set of possibilities (like the figure above).

There are two common approaches of static code analysis: intra-procedural static analysis and inter-procedural static analysis.

inter-procedural	**intra-procedural**
Follows/expands all function calls to evaluate those functions	runs linearly through the function code to analyse (low runtime but state inference is less concrete)
High runtime cost (potentially exponential run time)	low runtime but less potential state inferred
not scalable in many cases	scalable (less info though)

4.3 Branch Prediction

We discussed the pipeline prefetching data using the fetch/decode/execute model allowing data to be loaded into higher level caches and the pipeline to reduce access time. Then using **speculative execution**, the results from speculatively loaded instructions can be pre-computed (but hold the data back and don't commit until the branch is confirmed to be the correct branch.

Discussed CPU operation parallelization. This is called: ILP (Instruction Level Parallelizing). Unlike with loop parallelization the dependencies are able to be determined as they are all simply register addresses for the computations (if no registers in common for the operations they can be done in parallel, otherwise execute them sequentially).

We discussed branch prediction (to reduce time taken for loading as the instructions should already be in the pipeline (hopefully). Due to pipeline stalls/bubbles being costly in terms of computation time, we talked about the various approaches to branch prediction.

models of prediction:

1. **Back Edge Taken**: If the loaded data is in a loop, reload the code for loop at the jump back instruction., advantages: simple, usually accurate for loops but not always (wrong if that was the last iteration of the loop).
2. **Last Path Taken**: Just load the data the way we did last time. This has benefits in terms of if statements. If in a loop that predictably goes down the same path, like the Back Edge Taken model though, it tends to fail when ends of loops come

Modern Branch Predictors need higher accuracies than the 85-95% accuracy these simple models offer. As a result, more sophisticated models with accuracies around 99.9% are currently used.
3.a. **Perceptron Branch Predictor** (used in Samsung devices)
3.b. **PHT (Pattern History Table)** - used in Intel

4.4 Dynamic Memory Allocation

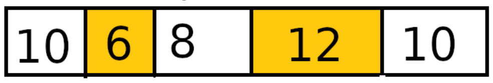


Figure 4.4: memory fragment, white is free

We covered dynamic memory, which is also called "heap", where we allocate objects and free them when we are done with them. We need to manage them somehow and try to reuse the freed blocks when possible. An easy way is to maintain a linked list called free list, where each node is a free block.

Which block should we reuse when new allocation happens? There are two different strategies we covered on this subject:
1. **First fit** will allocate the object to the first big enough block in free list. This method is easy and fast, but space inefficient. In Figure 4.4, we will use the first block if we allocate an object of size 8.
2. **Best fit**: will allocate the object to the free-block that has the smallest possible size that will be capable of fitting the object. In Figure 4.4, we will use the third block if we allocate an object of size 8. It is easy

to notice that we may need to iterate through the entire free list, which costs potential O(n) runtime, so it is time inefficient. Fragmentation can actually make this worse memory use than first fit even.

There is also another concern: When the memory is broken into small pieces, it is hard to reuse any of them. This is called fragmentation. We will talk more about memory management and garbage collection in the next lecture.