Considering the performance implications of converting recursion to an explicit stack

devblogs.microsoft.com/oldnewthing/20180927-00

September 27, 2018

Raymond Chen

Suppose you have a recursive algorithm, such as in-order tree walking.

```
void add_to_each_node_recursive(Node* root, int extra)
{
 if (!root) return;
 root->value += extra;
 add_to_each_node_recursive(root->left, extra);
 add_to_each_node_recursive(root->right, extra);
}
```

Suppose further that the stack is large enough and the data set small enough that stack overflow is not a problem. Will converting the algorithm to a non-recursive algorithm with an explicit stack provide any performance improvement?

For expository convenience, I will call this second version “iterative”.

```
void add_to_each_node_iterative(Node* root, int extra)
{
 std::stack<Node*> stack;
 for (;;) {
  while (root) {
   root->value += extra;
   if (root->right) stack.push(root->right);
   root = root->left;
  }
  if (stack.empty()) return;
  root = stack.pop();
 }
}
```

Determining which version will have better performance is complicated.

The recursive version will take advantage of a hardware stack, assuming the processor has one. So pushing and popping activation frames may be faster than manually manipulating a stack data structure.

On the other hand, you have function call overhead and ABI constraints. The compiler has to build a proper ABI-compliant stack frame, and it will almost certainly save and restore additional registers as part of that stack frame. Parameters that are constant through the entire operation such as the `extra` parameter will still need to be passed to each recursive call. The ABI may require things like home space and stack pointer alignment, which will add to the size of the frame.

The iterative version reuses the same `extra` variable, so it doesn't have to push it onto the explicit stack. All that goes onto the explicit stack is a single pointer, namely the point at which to resume the tree walk after exploring the left branch.

Lower memory requirements point toward the iterative version because that increases the amount of useful data that will fit inside the L1 cache.

The recursive version uses the hardware procedure call mechanism, which means that it gets to take advantage of any return address predictor. However, those predictors tend to max out at around 16 to 32 calls, so if your structure is deeper than that, you are going to overflow the return address predictor.

The iterative version doesn't use the hardware procedure call mechanism at all. Instead, it relies on the branch predictor. So now it depends on how you structured your code: Did you set things up so that the branches usually go the same way? For example, maybe the branch is always taken, except for the final iteration.

The recursive version may consume a lot of stack, although we're assuming that the stack is large enough to accommodate the worst case. As the function runs, the stack grows to accommodate the recursion, and when the function returns, that stack growth is not rolled back. The extra stack has been allocated, and it won't go away until the thread exits. Sure, it'll get paged out eventually, and will even be reused if you call another deeply recursive function. But that reuse is going to have to page the stack back in.

The iterative version creates a temporary stack, uses it for the purpose of the algorithm, and then destroys it. So there is no lingering effect of the iterative version.

Okay, well, you did have to allocate the memory from the heap, and the heap may have had to expand, but when you destroy the stack, that memory becomes available to the rest of the program, so it's not lost forever.

But that introduces a sticking point: There's a lot of code hiding inside the `stack.push` and `stack.pop` operations. You have the potential for heap allocations and deallocations, which means incursions into your precious branch prediction and cache (both code and data). It also reduces optimization opportunities for the compiler because a call into the heap is going to destroy all the volatile registers.

So which way do the scales tip?

As with most issues of performance, the only way to know is to measure. There's no single answer. You'll have to run tests on your specific workload to determine which works better for you.

Bonus chatter: Don't forget the engineering cost. The recursive version is usually easier to write, understand, and debug.

Raymond Chen

Follow