

@llvm.parallel_alloca

Julius Ikkala and Pekka Jääskeläinen / Tampere University

Hoisting of `alloca`

- `alloca` instructions in LLVM are generally hoistable
 - There does exist a non-hoistable `alloca`, but only when `getArraySize()` is non-constant :(
 - Hoisted by at least inliner & Clang's `CodeGenFunction::CreateTempAlloca()`
- If you declare a stack variable in a loop, it actually gets hoisted to the start of the function
 - Usually, this is not a problem, as the lifetime of the variable doesn't overlap between loop iterations (serial loop: previous iteration is done before next one runs)

Parallel problem

- However, in a vectorized loop, multiple iterations may run in parallel
 - The memory of a hoisted alloca is shared between iterations -> unexpected memory dependency
- LoopAccessAnalysis notices the alloca-introduced memory dependency and prevents vectorization
- How about parallel loops?
 - Annotated with `@llvm.loop.parallel_accesses`
 - These skip most of LoopAccessAnalysis and assume no memory dependencies between iterations

Trainwreck

- Inadvertent race condition causes miscompilations:

```
2 void foo(int *A, int n) {  
3     #pragma clang loop vectorize(assume_safety)  
4     for (int i = 0; i < n; i++) {  
5         int t[32];  
6         t[i] = i;  
7         t[0] = 0;  
8         A[i] = t[i];  
9     }  
10 }
```

```
2 void foo(int *A, int n) {  
3     int t[32];  
4     #pragma clang loop vectorize(assume_safety)  
5     for (int i = 0; i < n; i++) {  
6         t[i] = i;  
7         t[0] = 0;  
8         A[i] = t[i];  
9     }  
10 }
```

- Accesses to `t` seem safe, but `t` gets hoisted and introduces a new race condition which the programmer can't avoid!
 - `alloca` in a parallel loop isn't safely hoistable!

Fix attempt #1: detect bad accesses

- If a load / store instruction in a parallel loop accesses an `alloca`, don't consider the loop parallel anymore
- Bad in a few ways:
 - Only prevents vectorization – fixes miscompile, but doesn't fix performance
 - Unreliable, we can't actually safely detect if a pointer is related to an `alloca`
 - Only treats symptoms, not the root cause (of hoisting `alloca` without downgrading into a sequential loop)
- Ideally, we would avoid hoisting `alloca` and also make it vectorize somehow

Fix attempt #2: @llvm.parallel_alloca

- We can't demand that all passes are aware of parallel metadata, so we can't make `alloca` not get hoisted in parallel loops
- But we can add a new intrinsic that isn't hoistable and is vectorizable

`<ptr x n> @llvm.parallel_alloca(i64 size, i64 align)`

- Can start with a single ptr return value, gets vectorized to `<n>` based on what vectorization factor the loop vectorizer wants
- "Privatization": each lane gets a unique pointer with at least `size` bytes and correct `alignment`
- Address space is determined from return pointer type

Transforms/Utils/LowerParallelAlloca.h/cpp

- The implementation for the lowering is a single function:

```
Value* lowerParallelAlloca(IntrinsicInst *ParallelAlloca);
```

- **If** `ptr @llvm.parallel_alloca(s, a):`
 - **Generates** `alloca [s x i8], align a`
 - **Hoists** the `alloca` to the entry block
- **If** `<ptr x n> @llvm.parallel_alloca(s, a):`
 - `K = round_to_next_multiple_of(s, a)`
 - `%0 = alloca [(K * n) x i8], align a`
 - `%1 = getelementptr inbounds i8, ptr %0, <0, K, 2K, ...>`
 - **Replace** all uses with `%1` – each lane gets unique pointer with `K` bytes

Mem2Reg support

- We still want Mem2Reg to work with `parallel_alloca`, because vector registers naturally "privatize" the allocation
- Luckily, there's a surprisingly easy logic to do this:
 - If the `parallel_alloca` was lowered now, would the resulting `alloca` be register-promotable?
 - If so, lower it now and proceed just like with every other `alloca`
 - Calls `lowerParallelAlloca`

Inliner support

- If the call-to-be-inlined has `@llvm.access.group`, it operates in a parallel context
 - Just upgrade `alloca` to `parallel_alloca` while inlining and don't hoist
- It's OK for the inliner to have to be aware of parallel metadata:
 - It already is aware so that it can propagate `@llvm.access.group` from function call to inlined loads/stores
 - (there wouldn't be a miscompile if it didn't do that – a fully parallel-unaware inliner would also work correctly and make the loop serial because of unannotated loads/stores)

LoopVectorize support

- Already tries to vectorize intrinsic calls with `VPWidenIntrinsicRecipe`
- `@llvm.parallel_alloca` happily vectorizes
 - Arguments are uniform (size and alignment)
 - Return value is vectorized: `ptr => <ptr x VF>`
- Small special handling: if the vectorized intrinsic is `@llvm.parallel_alloca`, also call `lowerParallelAlloca` immediately
 - Not strictly necessary, but can allow for later optimization passes to operate on the lowered instructions before `ISel`

PreSelIntrinsicLowering fallback

- If no prior pass handled `@llvm.parallel_alloca`, it is finally lowered in `PreSelIntrinsicLowering`
 - Typically, this means that it lowers into a normal `alloca`
 - Because there can not be any vectorization passes after this, parallel metadata is no longer used and we can safely disregard it
- > Lowering to normal `alloca` and hoisting it is safe
- (calls `lowerParallelAlloca`)