Lean, Mean Coroutines: Hack Asynchronous Optimizations

Alexander Brassel, brasselsprouts@fb.com, Facebook

***Abstract*—Hack [2] is the language of choice for a large portion of Facebook's server-focused codebase. Hack programs make heavy use of asynchronous programming, using the language's async keyword to compile functions as suspendable coroutines. This paper explores several different algorithms to reduce how much memory coroutines consume at runtime and the size of their binaries.**

I. Introduction

The Hack language, a fork of PHP, was developed at Facebook and made public in 2014. It refines the PHP language in several ways, notably with the introduction of gradual typing, a typing system where types are implicit where inferrable, and explicit where there are not. Designed for use at Facebook, it now has mature asynchronous programming capabilities.

Asynchronous programming is a form of concurrent programming in which instructions or units of work are considered tasks, and are executed in a potentially non-linear, or concurrent fashion. There are a variety of ways to enable asynchronous programming for a language, but Hack chooses to represent coroutine calls using a spill-and-restore model. Coroutines are asynchronous function calls, and a suspendpoint is an instruction that is a call to a coroutine. They are referred to as suspendpoints because they require suspending the state of the function until resumed. In this model, when a suspendpoint is encountered, the set of variables that are in scope before the call and will be needed to be in scope after the call are computed (denoted the "live" instructions). These variables are stored in an auxiliary, compiler-generated datastructure in an act called "spilling". Later, when the coroutine has completed and the control flow is ready to be resumed, the auxiliary state for the coroutine is examined and the stored variables are read back into from their fields.

Only a single suspendpoint is spilled at any one time. Therefore, a single auxiliary storage datastructure is allocated with sufficient size to store any of the suspendpoints' live variables. Prior to the optimizations described in this paper, the auxiliary datastructure's fields were allocated in a simple, naive fashion: each suspendpoint adds enough fields for all of its live variables, regardless of the fields already present. Then, for each suspendpoint, a sequence of instructions are issued to store and subsequently load each suspendpoints' variables.

Given the large size of the Facebook Hack codebase, one of the primary objectives of the Hack compiler is to reduce the size of the executable machine code wherever possible. Here, this can be realized by reducing the number of "store" instructions issued for the suspendpoints in a function. A secondary objective is reducing the heap storage used by the suspended coroutines. This can be done by increasing the number of fields suspendpoints share when they are spilled.

A. Motivating Example

The code in appendix D demonstrates one of the failure modes of naively allocating fields for coroutines. The `big_wait_handle` function contains ten suspendpoints (the statements containing the keyword `live`). At each of these suspendpoints, any variable that will be reused after the suspend instruction (aka a `live` instruction) must be kept in storage. There are 11 fields at each suspendpoint, since all of the non-n variables are needed for the call to `sink` afterwards, and n is needed throughout the entire lifetime of the function. The naive approach allocates $11 * 10$ fields, even though only at most 11 of those fields are ever used at any one time.

II. Approach

A. Less naive

Following a philosophy of incremental development, the first iteration of asynchronous optimization is a simple greedy algorithm that will seek to reduce the number of fields used by an asynchronous functions' coroutines. There is no constraint that stipulates that the spillage of a variable must be consistent across suspendpoints, so simply allocate the minimum number of fields for each type represented in a given suspendpoint. Then, since the auxiliary state is global with respect to the function, allocate the maximum of these minimums per type. Then, the minimum number of fields to spill all suspendpoints will have been used. In practice, this can be done by keeping a type mapped pool of fields and lazily creating a new value whenever a pool is empty for a particular type. This approach performs very well in terms of memory consumption, but requires each suspendpoint to handle spilling their own instructions to their own fields. An excerpt for the source code of this approach is attached in appendix A.

1) Improvements on the previous example: Referring again to the Hack code sample from appendix D, at the first suspendpoint in live 5, the greedy algorithm sequentially allocates fields for n to $a9$. The fields are, somewhat arbitrarily, named $f0$, to $f10$. Then, at the next suspendpoint, the fields from the previous iteration ($f0$ through $f10$) are available, and since the types of bX are compatible with the types of aX (and for that matter n), $f0$ through $f10$ are reused for $b0$ through $b10$.

Mat Hostetter, Facebook
Edwin Smith, Facebook
Jan-Willem Maessen, Facebook

At the conclusion of the greedy algorithm, only 11 fields are used. This is a marked improvement over the $10*11$ fields from before. Even more remarkably is that the number of fields used is not directly tied to the number of times a function suspends.

B. Coloring

The second iteration of async optimization focuses on spilling deterministically at the cost of optimizing the heap usage of the program at runtime. Doing so unlocks optimizations for reducing the size of the emitted machine code. To this end, spilling properly can be recast as a graph coloring problem, much like in register allocation [3]. The vertices are the instructions, and each suspendpoints' live instructions form a clique. Then, the field assignments are colors. This satisfies the constraint that no two variables share the same field at the same suspendpoint, while still giving each instruction exactly one color. Therefore, if the coloring is done well, we can save on binary size while also still saving on heap usage at runtime.

Constructing an explicit graph with the density necessary to represent this problem is too computationally expensive, so an implicit construction is used instead. In this approach, we take heavy advantage of the SSA structure of the Hacknative compiler. In SSA architecture, it is safe to assume that a variable is declared exactly once. Dominator trees are a tree of SSA blocks such that every child block can only be reached via control flow through the parent. This, combined with dominator trees, allows one to reason about unique variables over their lifetime (in multiple branches of the tree) with no risk of lifetime holes. A lifetime hole refers to the phenomenon where, when iterating over a function, a variable disappears from scope and then reappears later.

The coloring algorithm can be broadly divided into two parts. The first is a pre-processing DFS postorder walk of the dominator tree which constructs several useful invariants for the forwards pass. The second is a RPO walk of the dominator tree that handles coloring using the invariants from the first pass. The behavior of the two passes will be described in reverse order of their chronological appearance in the algorithm - first the forwards pass, and then the pre-processing page. This is despite the fact that the pre-processing pass occurs first. The algorithm is presented in this order so as to justify the information that the pre-processing pass will collect.

1) Forwards: During the forwards phase, the first time a variable is encountered, we can be certain that this is the block in which it is initialized, due to SSA. It is critical that a variable takes a field as a soon as it is declared, or else branches in control flow might lead to conflicting spillages. To this end, a set is built during the initial pre-processing phase containing all variables that will ever be spilled throughout the lifetime of the function. If the variable is present in the set, it gets a field when it is first encountered. There is a global list of fields maintained, and at the beginning of each block, the global list is copied and filtered so that variables that are live at block entry have exclusive claim to their fields. The pre-processing pass also handles creating the list of live variables at block entry.

To pick a field for a variable, we pick the next field from the freelist of the correct type. If there are no fields available, a new one is created. Whichever option is pursued, that field is marked permanently (colored) for the variable.

The next tricky bit comes when a suspendpoint is hit in a block. Pre-processing (whose purpose is becoming increasingly apparent) also maintains a list of variables that will be spilled again after a suspendpoint. This is distinct from the live set of suspendpoint variables, because that is only the list of variables that will be *used* again. If an instruction is going to be used again, but not spilled, it does not need to keep a claim on a field, and other instructions are free to use that field.

2) Backwards: The pre0processing phase iterates in DFS postorder through the dominator tree and in reverse instruction order through each block. It builds a set of ever spilled variables, a mapping from block ids to their live variables at entry, and a list of variables that will be spilled again after a suspendpoint.

The algorithm can be built up inductively: consider a particular block. The variables that will be spilled in any of its children can be constructed as the union of the variables that are live at the entry of each child block. After that, iterate backwards through the block. Any variable encountered in this block can be removed from the list of variables that will be spilled, because no parent block will care about variables that are declared after its body. However, the variable should be noted as one of the spilled instructions at this point.

On the other hand, if a suspendpoint is reached, then the variables that will be spilled after the suspendpoint are precisely the variables that will be spilled again at the point the suspendpoint is encountered. After the suspendpoint, the list of variables that will be spilled again is the set of live variables at the suspendpoint. Once all instructions in the block are iterated through, any remaining variables in the "will be spilled later" set must be declared in one of the parent blocks and spilled later. Therefore, they are the variables that are live at exit from the parent block.

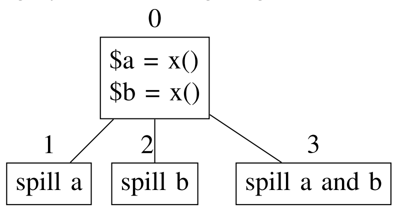
An excerpt of this approach can be found in A

3) Examples: The behavior of the algorithm in the appendix D example is the same, but not particularly interesting. Consider, instead, the inline code fragment produced below:

```
async function foo(int $n): Awaitable<int> {
  $a = x();
  $b = x();

  switch ($n) {
    case 0:
      await one();
      return $a;
    case 1:
      await one();
      return $b;
    case 2:
      await one();
      return $a + $b;
    default:
      return $n;
  }
}
```

The example above generates a dominator tree that includes roughly the following fragments.



During the pre-processing phase, block 3 begins iterating through its variables with an empty set of live variables, as it has no children. It encounters the first suspendpoint, which informs it that $a and $b will be spilled here. This suspendpoint is informed that no variables are live after it. Then, the tracked live variables are expanded to include $a and $b. The block finishes and notes that at its entry $a and $b are live. A similar process repeats on block 2 and block 1. Then, at block 0, the variables live at the end of the block are computed as the union of {{$a, $b}, {$a}, {$b}}, or {$a, $b}. Then, variables $b, and then $a are encountered in turn. Finally, at the beginning of the block, there are no instructions left, so the live variables at the beginning of the block become the empty set.

Next, during the forward iteration phase, $a and $b are spilled at block 0 into fields f0 and f1. They can't be spilled into the same field because $a is alive when $b is to be spilled. Next, at block 1, $b loses its reservation on f1 because it is not in the instructions live at the entry to the block. This would allow any new SSA variable declarations to reuse its field. A similar phenomenon happens in block 2. Finally, in block 3, both are still alive, so neither are freed.

C. Tail Merging

Finally, we can take advantage of consistent spilling per variable to reduce the emitted binary footprints. In order to do so, a few observations must be made. First, the order in which variables are spilled for a suspendpoint is unimportant. Second, all suspendpoints will jump to the same location (loosely speaking) after spilling their variables. Third, shared blocks of variables can be jumped to at very little extra cost. Given these facts, it is possible to tail merge the process of spilling each suspendpoint. Common shared subsets of live variables for each suspendpoint can be represented as a single block that spills all of the shared variables. The ends of these blocks of code can all jump to the same post-spillage destination, or, even better, other blocks of shared spillage code.

To provide an example, consider the following list of suspendpoints:

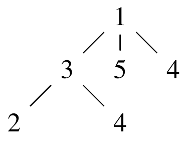
```
[{1, 2, 3}, {1, 3, 4}, {3, 1}, {1, 3},
 {1}, {1, 4}, {1, 5}]
```

At spill, each suspendpoint's live SSA variables need to be written to their fields. Because of our previous round of optimizations, each variable has a single, known field. The first of these anonymous suspendpoints can be spilled in the order 1, 2, 3. However, spilling commutes: it might just as well have been spilled in the order 3, 2, 1.

If we were to reorder these suspendpoints as follows:

```
[{1, 3, 2}, {1, 3, 4}, {1, 3}, {1, 3},
 {1}, {1, 4}, {1, 5}]
```

then it becomes apparent that they could all spill the "1" variable last. Indeed, a dedicated block can be created spilling just 1, and then each suspendpoint can jump to it after spilling all of their other variables first. This reduces the number of variables spilled 1 from 7 to 1. This process can be recursively repeated too; many of these variables spill 3, and then spill 1. They could spill any non-3 and non-1 variable, and then jump to a block that spills 3, and then spills 1. This approach inductively builds up to a forest mapping blocks of spilled instructions via jumps.



Inductively, this can be represented as a forest of shared spillage blocks where each block jumps to another block until exit from the spillage routine. If done correctly, each suspendpoint's set of variables to spill will be represented by a branch in the forest. Even better, the compiler is smart enough to compact sequences of blocks in the forest which are only jumped to by at most one other block. This makes it possible to peel off a single instruction at a time without increasing the number of blocks at runtime.

This problem is modeled by treating the input suspendpoints as a list of sets. Some variable is picked and the sets are divided into those that contain the variable and those that do not, much like in ZDD [1]. Then, the suspendpoints that do contain the variable are stripped of it and a block is created with that variable being spilled and a jump to that blocks parent. Then, a forest is constructed recursively out of the list of suspendpoints that had the instruction, and the result is set as the children of the new block. On the other hand, the suspendpoints that did not have this variable are recursed on and their result is set as the child of the parent block. By

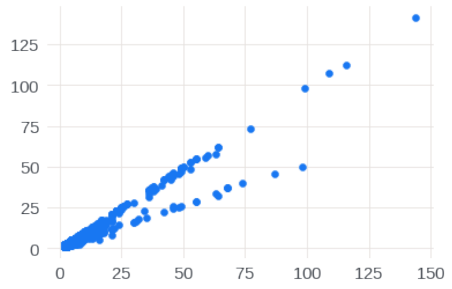


Fig. 1. Ratio between naive and greedy allocation as a function of suspendpoint count

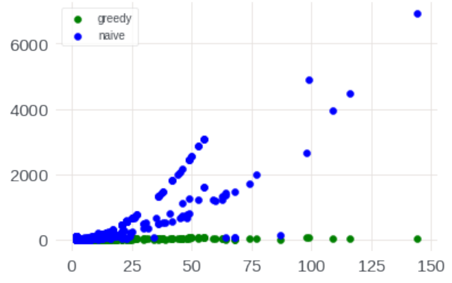


Fig. 2. Field count correlated against number of suspendpoints

seeding this recursion with the block to be jumped to after the spillage is complete, the recursion will seamlessly assign the jump for any root block in the forest to be the post-spillage destination.

The next problem is to identify where a suspendpoint should jump to in this forest. Whenever a suspendpoint is empty during construction of the forest, the current block id is its entry point. This is because induction ensures that the suspendpoint will have been emptied when all of its variables are in the current path to the root.

There are many different ways to order picking the next instruction, but the implementation provided here uses a simple one - greedily picking the most commonly occuring instruction. An implementation can be found in Appendix C.

III. Analysis

The greedy approach (Appendix A) performs admirably well against naive. As shown in Fig 1, the percent allocation savings balloon as the number of suspendpoints increase.

To drive the point home, the amount of memory used hardly grows with an increase in suspendpoints (Fig 2).

The ratio between greedy and naive field allocation has a harmonic mean of 41.4%, when accounting for the suspendpoint size distribution. Which is worth noting, since over half of the functions do not have any await points (Fig 3). This implies that asynchronous functions that spill save on average 58.2% of their potential field allocations.

Graph coloring performs nearly identically to the greedy approach even though it is technically suboptimal (Fig 4).

As shown on the graph, there are cases in which graph coloring produces suboptimal results, but in practice they are uncommon. Graph coloring scales well with instructions (Fig 5) and suspendpoints (Fig 6).

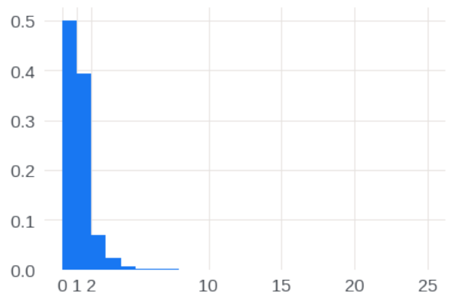


Fig. 3. Cumulative histogram of suspendpoint counts.

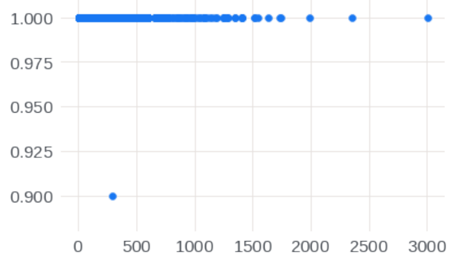


Fig. 4. Optimality ratio between greedy and graph coloring.

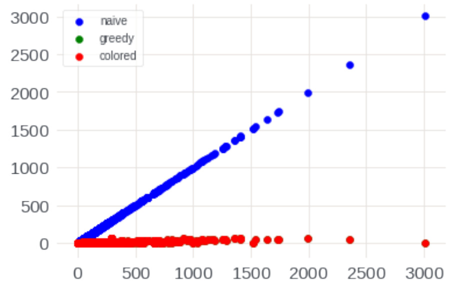


Fig. 5. Correlation between fields and variables

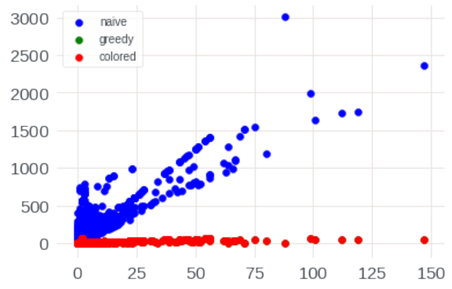


Fig. 6. Correlation between fields and suspendpoints

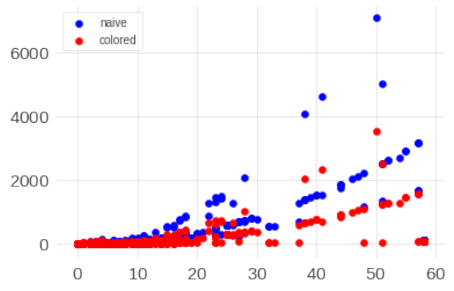


Fig. 7. Correlation between instructions to spill and instructions used to spill

As an interesting experiment, I correlated the number of variables that were to be spilled against the number of variables used to spill them for both the naive/greedy approaches and the colored algorithm (Fig 7). There appear to be three principle category of asynchronous functions, but regardless of the category, the coloring approach saw significant constant factor wins in footprint. The harmonic mean of the field allocation before and after for coloring was 41.4% nearly the same as greedy, and the binary footprint average savings were 22.1%.

The total field usage for the large code sample was 120,186 fields, vs 322,704 fields for a savings of 62.76%. The total number of spillage instructions before was 322,704, but graph coloring reduced them to 210,625, a savings of 34.73%.

IV. Conclusion

Greedily picking fields in and of itself represents a pretty huge memory win over naively allocating fields on spill. Roughly speaking, it causes the the number of fields needed to not scale with the number of suspendpoints. However, graph coloring nets most of the wins of greedily picking fields in practice while also significantly reducing the footprint of the binary. These are promising results for the optimization potential of the Facebook (and wider Hack) codebase.

V. Future Work

The current implementation of tail merging is not as efficient as it could be and contains several code smells. It needs to be refined in order to speed up compilation speeds. Furthermore, the work needs to be benchmarked a bit more extensively, possibly on the entire Facebook Hack codebase.

Acknowledgment

I would like to thank Mat Hostetter, Edwin Smith, and Jan-Willem Maessen for volunteering their time, guidance, and expertise on this project. I quite literally could not have done it without them.

I'd also like to thank Raj Yengisetty for being incredibly supportive of me during the course of this project.

References

[1] Shin-ichi Minato. 1993. Zero-suppressed BDDs for set manipulation in combinatorial problems. In Proceedings of the 30th international Design Automation Conference (DAC '93). Association for Computing Machinery, New York, NY, USA, 272–277. DOI:https://doi.org/10.1145/157485.164890
[2] Hack. (2021). Retrieved May 13, 2021, from https://hacklang.org/
[3] Register Allocation. (2021, Spring). Retrieved May 13, 2021, from http://web.cecs.pdx.edu/ mperkows/temp/register-allocation.pdf

APPENDIX A
GREEDY

```rust

#[derive(Debug)]
struct SpilledInstr {
    // map sp -> fields
    fields: InstrIdMap<FieldIndex>,
    // This is the original, desired ty; the TyId for this InstrId in COMMON
    // may be wider, for func params whose types we had to change.
    ty: TyId,
    local_decl: InstrId,
}

type InstrIdInfoMap = IndexMap<InstrId, SpilledInstr, BuildIdHasher<u32>>;

#[derive(Debug)]
struct FieldAllocation {
    // All physical fields.
    fields: IdVec<FieldIndex, (FieldPath, Field)>,

    // Information about each spilled InstrId.
    iid_info: InstrIdInfoMap,
}

fn allocate_fields(
    func: &Func,
    spoints: &[Suspendpoint],
    coro_ty: TyId,
    tag_env: &TagEnv,
) -> FieldAllocation {
    let tenv = tag_env.tenv;

    let mut iid_info: InstrIdInfoMap = Default::default();
    let mut type_to_fields: TyIdMap<Vec<FieldIndex>> = TyIdMap::default();
    let mut fields: IdVec<FieldIndex, (InstrPName, Field)> = IdVec::default();

    for sp in spoints.iter() {
        // `positions` tracks the next field to be used for each type.
        let mut positions: TyIdMap<usize> = Default::default();
        for &iid in sp.live.iter() {
            let pname = func.instrs[iid].pname();
            let ty = func.instr_ty(iid, tenv);
            let defaultable_ty = types::defaultable_supertype(ty, tenv);
            let field_ty = tag_env.widen_to_repr_ty(defaultable_ty);

            let spill = iid_info.entry(iid).or_insert_with(|| SpilledInstr {
                fields: Default::default(),
                ty,
                // local_decl will be filled in later.
                local_decl: InstrId::NONE,
            });

            let field_ids = type_to_fields.entry(field_ty).or_default();

            let position = positions.entry(field_ty).or_default();

            let field_id = match field_ids.get(*position) {
                Some(&field_id) => {
                    if fields[field_id].0 != pname {
                        fields[field_id].0 = InstrPName::EMPTY;
                    }

                    field_id
                }
                None => {
                    // In this case, we need another field to spill this instruction.
                    let field = Field {
                        ty: field_ty,
                        mutability: Mutability::Assignable,
                    };

                    let index = FieldIndex::from_usize(fields.len());
                    fields.push((pname, field));
                    field_ids.push(index);
                    index
                }
            };

            // Having used this field, advance to the next one.
            *position += 1;

            spill.fields.insert(sp.iid, field_id);
        }
    }

    // `fields` currently tracks uniqueness instead of an explicit path.
    // Transform `fields` appropriately for `FieldAllocation`.
    let fields = name_fields(coro_ty, fields);

    FieldAllocation { fields, iid_info }
}
```

As a brief explanation, the `allocate_fields` method is responsible for creating the fields for the spill state as well as creating a map from instruction to the field it will spill to at each suspendpoint. In this algorithm the same instruction may spill to different fields at different suspendpoints, so the `SpilledInstr` contains a mapping from suspendpoint (an instruction id) to an index in to the fields.

It's also worth noting that the routine practices type-widening - buckets fit smaller types in them too.

Appendix B
Coloring

```
fn allocate_fields(
    func: &Func,
    spoints: &[Suspendpoint],
    coro_ty: TyId,
    tag_env: &TagEnv,
) -> FieldAllocation {
    let tenv = tag_env.tenv;
    // We want to look up in the set of spillpoints, so keep a set of ids ready.
    let spoint_ids = spoints
        .iter()
        .map(|spoint| (spoint.iid, spoint))
        .collect::<InstrIdMap<_>>();


    // Explicitly build the DOM tree.
    let pred = Rc::new(predecessors::compute_predecessors(func));
    let dom = dominator::compute_dominator_parents(func, &pred);
    let dominator_children = dom.children();

    let mut alive_before_block: BlockIdMap<InstrIdSet> =
        BlockIdMap::with_capacity_and_hasher(func.blocks.len(), Default::default());
    let mut declares_spilled_fields: HashSet<BlockId> = HashSet::with_capacity(func.blocks.l
    let mut ever_spilled = InstrIdSet::default();
    let mut killed_after_sp: InstrIdMap<Vec<InstrId>> =
        InstrIdMap::with_capacity_and_hasher(spoints.len(), Default::default());

    for bid in func.block_ids().rev() {

        let mut alive: InstrIdSet = dominator_children[bid]
            .iter()
            .map(|&bid| alive_before_block[&bid].iter().copied())
            .flatten()
            .collect();
        let mut does_declare_spilled_fields = false;

        for iid in func.blocks[bid].iids().rev() {
            if let Some(Suspendpoint { live, .. }) = spoint_ids.get(&iid) {
                let now_live = live.iter().copied();
                // Add all spilled instrs to the global list of ever-spilled instrs.
                ever_spilled.extend(now_live.clone());
                // Any instr that is no longer alive after this sp should be marked as kille
                killed_after_sp.insert(
                    iid,
                    now_live
                        .clone()
                        .filter(|iid| !alive.contains(&iid))
                        .collect(),
                );
                alive = now_live.collect();
            }

            if alive.remove(&iid) {
                does_declare_spilled_fields = true;
            }
        }
        alive_before_block.insert(bid, alive);
        if does_declare_spilled_fields {
            declares_spilled_fields.insert(bid);
        }
    }

    struct InstrInfo {
        pname: InstrPName,
        ty: TyId,
        field_ty: TyId,
    }

    let mut iid_meta: InstrIdMap<InstrInfo> = ever_spilled
        .into_iter()
        .map(|iid| {
            let ty = func.instr_ty(iid, tenv);
            let defaultable_ty = types::defaultable_supertype(ty, tenv);
            let field_ty = tag_env.widen_to_repr_ty(defaultable_ty);
            let pname = func.instrs[iid].pname();

            (
                iid,
                InstrInfo {
                    pname,
                    ty,
                    field_ty,
                },
            )
        })
        .collect();

    // Build data to return.
    let mut fields: IdVec<FieldIndex, (InstrPName, Field)> = Default::default();
    let mut iid_info: InstrIdInfoMap =
        InstrIdInfoMap::with_capacity_and_hasher(iid_meta.len(), Default::default());
    let mut pool: TyIdMap<Vec<FieldIndex>> = Default::default();

    for bid in func.block_ids() {
        if !declares_spilled_fields.contains(&bid) {
            continue;
        }

        // Go from the used instrs to the used fields.
        let fields_in_use: HashSet<FieldIndex> = alive_before_block[&bid]
            .iter()
            .map(|iid| iid_info[iid].field)
            .collect();

        // For each type, filter out fields that are in use.
        let mut freelist: TyIdMap<BinaryHeap<_>> = pool
            .iter()
            .map(|(&tyid, fields)| {
                (
                    tyid,
                    fields
                        .iter()
                        .filter(|field_id| !fields_in_use.contains(field_id))
                        .map(|&field_id| Reverse(field_id))
                        .collect(),
                )
            })
            .collect();

        for iid in func.blocks[bid].iids() {
            // Kill instrucions immediately after their last spill.
            if spoint_ids.contains_key(&iid) {
                for dead_iid in &killed_after_sp[&iid] {
                    let freelist = freelist.entry(iid_meta[dead_iid].field_ty).or_default();
                    freelist.push(Reverse(iid_info[dead_iid].field));
                }
            }

            // Allocate instructions as they are found (if they'll ever be spilled).
            if let Some(InstrInfo {
                ty,
                field_ty,
                pname,
            }) = iid_meta.remove(&iid)
            {
                let field = match freelist.entry(field_ty).or_default().pop() {
                    Some(Reverse(iid)) => iid,
                    None => {
                        // If there is no field available, make a new one.
                        let field = Field {
                            ty: field_ty,
                            mutability: Mutability::Assignable,
                        };

                        let index = FieldIndex::from_usize(fields.len());
                        fields.push((pname, field));

                        pool.entry(field_ty).or_default().push(index);

                        index
                    }
                };

                // Mark name as empty if there's a naming conflict for a field.
                if fields[field].0 != pname {
                    fields[field].0 = InstrPName::EMPTY;
                }

                iid_info.insert(
                    iid,
                    SpilledInstr {
                        field,
                        ty,
                        // local_decl will be filled in later.
                        local_decl: InstrId::NONE,
                    },
                );
            }
        }
    }

    let mut uniq: HashSet<FieldIndex> = Default::default();
    assert!(spoints.iter().all(|spoint| {
        uniq.clear();
        spoint
            .live
            .iter()
            .all(|iid| uniq.insert(iid_info[iid].field))
    }));

    // `fields` currently tracks uniqueness instead of an explicit path.
    // Transform `fields` appropriately for `FieldAllocation`.
    let fields = name_fields(coro_ty, fields);

    FieldAllocation { fields, iid_info }
}
```

Appendix C
Merge

```rust
fn block_walk<MakeBlock>(
    exit: BlockId,
    spoints: InstrIdMap<InstrIdIndexSet>,
    mut make_block: MakeBlock,
) -> InstrIdMap<BlockId>
where
    //                (Parent , Instrs    ) -> New Block
    MakeBlock: FnMut(BlockId, &[InstrId]) -> BlockId,
{
    enum UniqueOrCount {
        Unique(InstrId),
        Count(usize),
    }
    impl std::ops::AddAssign<usize> for UniqueOrCount {
        fn add_assign(&mut self, rhs: usize) {
            match self {
                Unique(_) => {
                    *self = Count(rhs + 1);
                }
                Count(count) => {
                    *count += rhs;
                }
            }
        }
    }
    use UniqueOrCount::*;

    // Unfold the recursion.
    let mut stack = Vec::with_capacity(spoints.len());
    // Maintain a mapping from sp id -> entry block.
    let mut entries = InstrIdMap::with_capacity_and_hasher(
        spoints.len(), Default::default()
    );
    let mut spoints_unique_iids: InstrIdMap<Vec<InstrId>> =
        InstrIdMap::with_capacity_and_hasher(spoints.len(), Default::default());
    // Seed with the initial exit points and sets
    stack.push((exit, spoints));

    while let Some((mut parent, mut spoints)) = stack.pop() {
        let instrs_to_spoints = {
            let mut builder: InstrIdMap<UniqueOrCount> = Default::default();
            for (&sid, spilled_iids) in &spoints {
                for &iid in spilled_iids {
                    builder
                        .entry(iid)
                        .and_modify(|x| *x += 1)
                        .or_insert(Unique(sid));
                }
            }

            builder
        };

        let mut otherwise_most_common = None;
        let mut best_count = 0;
        let mut all_shared = Vec::default();

        for (spilled_iid, count) in instrs_to_spoints {
            match count {
                Unique(sid) => {
                    spoints.get_mut(&sid).unwrap().remove(&spilled_iid);
                    spoints_unique_iids
                        .entry(sid)
                        .or_default()
                        .push(spilled_iid);
                }
                Count(count) if count == spoints.len() => {
                    all_shared.push(spilled_iid);
                }
                Count(count) if best_count < count => {
                    best_count = count;
                    otherwise_most_common = Some(spilled_iid);
                }
                _ => {}
            }
        }
        if !all_shared.is_empty() {
            parent = make_block(parent, &all_shared);

            for spilled_iids in spoints.values_mut() {
                for iid in &all_shared {
                    spilled_iids.remove(iid);
                }
            }
        }
        spoints.retain(|&sid, spilled_iids| {
            if spilled_iids.is_empty() {
                let spill_bid = spoints_unique_iids
                    .remove(&sid)
                    .map_or(parent, |unique_iids| make_block(parent, &unique_iids));
                entries.insert(sid, spill_bid);
                false
            } else {
                true
            }
        });
        if let Some(most_common_iid) = otherwise_most_common {
            let mut have_nots = InstrIdMap::with_capacity_and_hasher(
                spoints.len() - best_count,
                Default::default(),
            );

            let bid = make_block(parent, &[most_common_iid]);

            let mut haves = spoints;
            haves.retain(|&sid, spilled_iids| {
                if spilled_iids.remove(&most_common_iid) {
                    true
                } else {
                    have_nots.insert(sid, std::mem::take(spilled_iids));
                    false
                }
            });

            stack.push((parent, have_nots));
            stack.push((bid, haves));
            }
        }

        entries
}
```

Appendix D
Example

```
async function big_wait_handle(int $n): Awaitable<int> {
   $a0 = x(); $a1 = x(); $a2 = x(); $a3 = x(); $a4 = x();
   $a5 = x(); $a6 = x(); $a7 = x(); $a8 = x(); $a9 = x();
   $n += await force_suspend();
   sink($a0, $a1, $a2, $a3, $a4, $a5, $a6, $a7, $a8, $a9);

   $b0 = x(); $b1 = x(); $b2 = x(); $b3 = x(); $b4 = x();
   $b5 = x(); $b6 = x(); $b7 = x(); $b8 = x(); $b9 = x();
   $n += await one();
   sink($b0, $b1, $b2, $b3, $b4, $b5, $b6, $b7, $b8, $b9);

   $c0 = x(); $c1 = x(); $c2 = x(); $c3 = x(); $c4 = x();
   $c5 = x(); $c6 = x(); $c7 = x(); $c8 = x(); $c9 = x();
   $n += await one();
   sink($c0, $c1, $c2, $c3, $c4, $c5, $c6, $c7, $c8, $c9);

   $d0 = x(); $d1 = x(); $d2 = x(); $d3 = x(); $d4 = x();
   $d5 = x(); $d6 = x(); $d7 = x(); $d8 = x(); $d9 = x();
   $n += await one();
   sink($d0, $d1, $d2, $d3, $d4, $d5, $d6, $d7, $d8, $d9);

   $e0 = x(); $e1 = x(); $e2 = x(); $e3 = x(); $e4 = x();
   $e5 = x(); $e6 = x(); $e7 = x(); $e8 = x(); $e9 = x();
   $n += await one();
   sink($e0, $e1, $e2, $e3, $e4, $e5, $e6, $e7, $e8, $e9);

   $f0 = x(); $f1 = x(); $f2 = x(); $f3 = x(); $f4 = x();
   $f5 = x(); $f6 = x(); $f7 = x(); $f8 = x(); $f9 = x();
   $n += await one();
   sink($f0, $f1, $f2, $f3, $f4, $f5, $f6, $f7, $f8, $f9);

   $g0 = x(); $g1 = x(); $g2 = x(); $g3 = x(); $g4 = x();
   $g5 = x(); $g6 = x(); $g7 = x(); $g8 = x(); $g9 = x();
   $n += await one();
   sink($g0, $g1, $g2, $g3, $g4, $g5, $g6, $g7, $g8, $g9);

   $h0 = x(); $h1 = x(); $h2 = x(); $h3 = x(); $h4 = x();
   $h5 = x(); $h6 = x(); $h7 = x(); $h8 = x(); $h9 = x();
   $n += await one();
   sink($h0, $h1, $h2, $h3, $h4, $h5, $h6, $h7, $h8, $h9);

   $i0 = x(); $i1 = x(); $i2 = x(); $i3 = x(); $i4 = x();
   $i5 = x(); $i6 = x(); $i7 = x(); $i8 = x(); $i9 = x();
   $n += await one();
   sink($i0, $i1, $i2, $i3, $i4, $i5, $i6, $i7, $i8, $i9);

   $j0 = x(); $j1 = x(); $j2 = x(); $j3 = x(); $j4 = x();
   $j5 = x(); $j6 = x(); $j7 = x(); $j8 = x(); $j9 = x();
   $n += await one();
```

```
  sink($j0, $j1, $j2, $j3, $j4, $j5, $j6, $j7, $j8, $j9);

  return $n;
}
```