Detecting Runtime Check Patterns and Applying Optimization in GVC0

Paulo Canelas *Sam Estep*

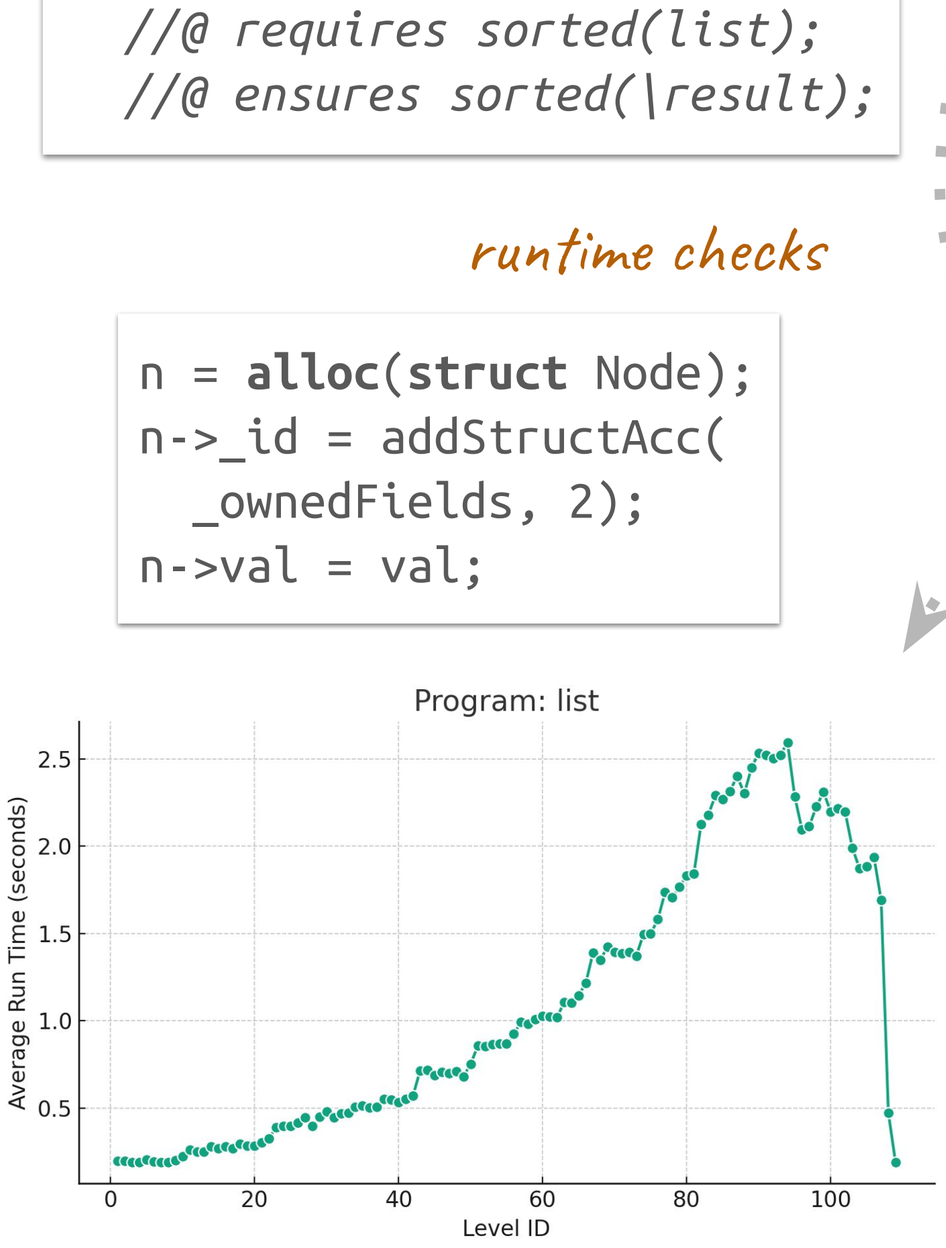
background: gradual verification

```
struct Node *list_insert(
  struct Node *list, int val)
  //@ requires sorted(list);
  //@ ensures sorted(\result);
```

runtime checks

```
n = alloc(struct Node);
n->_id = addStructAcc(
  _ownedFields, 2);
n->val = val;
```

Program: list



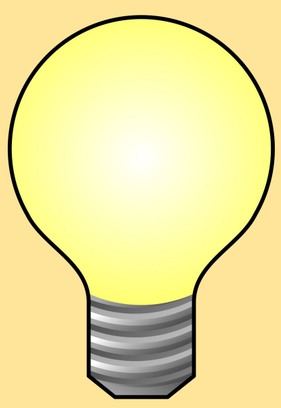
slow

why?

```
while (fields->contents[index] != NULL)
  if (!fields->contents[index]->deleted &&
      fields->contents[index]->_id == _id)
    return fields->contents[index];
  else
    index = (index + 1) % fields->capacity;
```

actually, C0 confuses LLVM in general

```
%28 = tail call i8*
  @c0_deref(i8* noundef %5)
  #6, !dbg !347
```



let's deduplicate C0 dereferences!

goal: move c0_deref calls out of loops

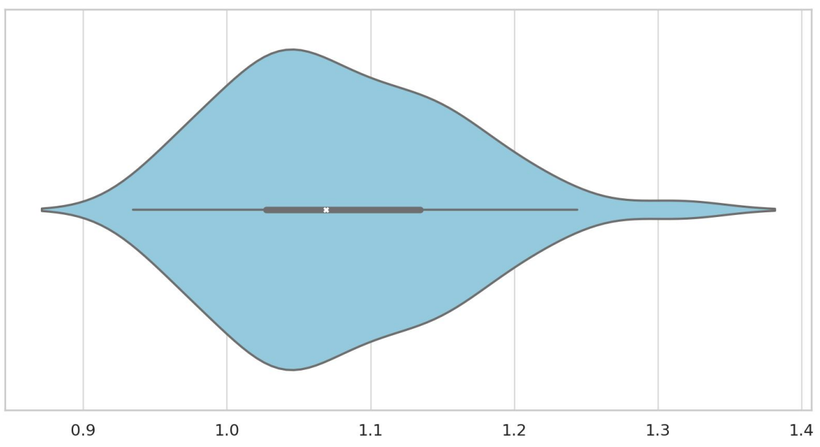
```
contents = fields->contents;
while (contents[index] != NULL)
  if (!contents[index]->deleted &&
      contents[index]->_id == _id)
    // ...
```

how do we know they're loop-invariant?

```
a = alloc(/* ... */);
b = a;
c = *a;
*b = /* ... */;
d = *a;
```

simple if the loop has no stores or calls

```
a = alloc(/* ... */);
b = a;
c = *a;
~~*b = /* ... */;~~
d = *a;
```



8% mean perf increase on slowest programs

✅ _c0_find	❌ _c0_initOwnedFields
✅ _c0_grow	❌ _c0_join
✅ _c0_newFieldArray	❌ _c0_list_insert
✅ _c0_printof	❌ _c0_sortedSegHelper

this covers the most impactful cases!