CS6241 Compiler Design - Project 1

Christopher Martin (gth773s) Dan Thayer (dthayer3) Esther Goh (egoh3)

Bounds check insertion

The `-abc-insert` pass inserts checks before each `GetElementPtr` (GEP) instruction for which a bound can be determined. The transformation looks like:

```
int A[n]
A[i] = x
```

→

```
  int A[n]
check1:
  if (i < n) goto "check2"
  exit(1)
check2:
  if (x ≥ 0) goto "gep"
  exit(1)
gep:
  A[i] = x
```

For each check (lower and upper) this requires calling `BasicBlock::splitBasicBlock(Instruction*)`, and inserting of `ICmpInst` and `CallInst` to the `exit` function.

If the array length is known at compile time, this is trivial because the length is included in the type of the array, which is readily available by inspecting the GEP instruction. For dynamically sized arrays, this can get much more complicated. This pass does a best-effort attempt to determine the lengths of dynamically sized arrays, and catches some cases. Particularly, it searches for a pattern of instructions which is typical of clang's output. In the following case, the length of `%8` is determined to be `%6`.

```
%6 = mul i64 4, %5 %7 = alloca i8, i64 %6, align 16 %8 = bitcast
i8* %7 to i32* %9 = getelementptr inbounds i32* %8, i64 10
```

The following example quits with an output of 1 when the bound check fails.

```
void foo(int n) {
  int a[n];
  a[5] = 3;
}
int main(int argc, char **argv) {
  foo(5);
  return 0;
}
```

Code size increase in phase 0 benchmarks:

benchmark	original size (bytes)	after check insertion	% increase
g721	18784	21320	13.5
gsm	110880	133736	20.6
jpeg	364604	411260	12.8
mpeg2	100864	126952	25.9
rawcaudio	3484	3524	1.1
rawdaudio	3476	3516	1.2

Bounds check removal

The `-abc-remove` pass implements an algorithm similar to ABCD. Pi nodes are inserted after each comparison. A pi node is represented as a PHINode with a single incoming assignment. We depend on the `BreakCriticalEdge` pass so that we can simply insert pi nodes at the top of each successor blocks rather than having to insert new blocks along edges. We also make use of `DominatorTree` analysis to determine where usages of values may safely be replaced by pi nodes (only in usages which are dominated by the pi).

For building the inequality graph, we only consider contraints of type C5 (comparisons).

The check removal phase does not work correctly due to some unknown logic error in our ABCD implementation. It is close, and we can generate the inequality graph, but it is flawed somehow.

Static opt results

For huffbench.c:

opt flags	bytecode size (bytes)	time (seconds)
unmodified	5548	16.39
-gvn	5512	10.92
-abc-insert	7840	16.01
-abc-insert -abc-remove	7312	16.08
-abc-insert -gvn	7412	12.39
-abc-insert -gvn -abc-remove	7080	12.38

The abc-insert pass gives a 41% increase in code size on `huffbench.c`.

GVN reduces the code size negligibly on the unmodified bytecode, but gives a 5% reduction after inserting array bound checks. This indicates that GVN is somewhat helpful in removing redundant checks.

The abc-remove pass shows a reduction in code size of 7%, or 4% after GVN. However, it gives no significant change in run time. We may need to take another look at the way in which branches are being removed, or consider whether the SplitCriticalEdges pass is significantly degrading performance.

In terms of code size alone, it seems that a combination for GVN and ABCD finds the most redundancy.

Dynamic opt results

Since the abc-removal pass did not successfully improve performance with static optimization, we can expect it to degrade performance in a dynamic compilation setting. Indeed, this is the case. Our benchmark `huffbench.c` runs in 16.00 seconds without abcd, and 17.19 seconds with abcd.