Redundant Array Bounds Check Removal

CS6241 Final Project

Ramyad Hadidi, Alireza Nazari

• Optimizations

A) Baseline

In the baseline implementation of bounds checking, we should add checks after each pointer access. This includes two steps of

i) Finding arrays and their bounds: In this part we implemented arrays with constant upper bound. This includes arrays which are defined normally *e.g. int a[C];* or arrays which are defined by *malloc* function.

ii)Finding memory accesses to recognized arrays: we traverse program and add checks after each in access to arrays in previous part. If index is constant, compiler generates and warning and points to the code line with an access beyond limit. If the index is variable, we implemented checks by adding a function calls to program which take the index, bottom and top limit of an array, check either top or lower limit and terminated the program if an violation of bound has happened. Note that *min* and *max* function prototypes look the same while their implementations are different and they only use either min or max.

Example:

> *%arrayidx59 = getelementptr inbounds [10 x double], [10 x double]* %arrayidx58, i32 0, i64 %idxprom, !dbg !170*
> *call void @__checkArrayBounds_min_i64_u64(i64 %idxprom, i64 0, i64 4)*
> *call void @__checkArrayBounds_max_i64_u64(i64 %idxprom, i64 0, i64 4)*
> *Runtime error:*
> **Array index 201 exceeds max of 200. Terminating.**

Implementation of function calls are in another source file which is compiled by clang beforehand and just links to test application object on compile time.

Baseline implementation has a big overhead as for the provided *STREAM* benchmark, it increases the executable file from 22.7 kB to 26.3 kB.

A) Local Elimination

This Optimization happens inside a basic block(Or modified CFG block as it's described in paper.). Our implementation handles any complexity of indices . It also finds and combines any two identical or subsumer checks.

```
1  int a[10];   int b[20]
2  b[j+f+1] = 0;
3  a[j+f+1]=0;
```

> *%arrayidx7 = getelementptr inbounds [10 x i32], [10 x i32]* %b, i32 0, i64 %idxprom6, !dbg !57*
> *call void @__checkArrayBounds_max_i64_u64(i64 %idxprom6, i64 0, i64 10)* → i+j+f<10*
> *call void @__checkArrayBounds_min_i64_u64(i64 %idxprom6, i64 0, i64 10)* → i+j+f>0*

Local elimination decreases size of STREAM executable to 26.0 kB.

B) Global Elimination

Global elimination has two sub algorithms to *i)Modify* and *ii)Eliminate redundancy* across CFG. Both algorithms are implemented as described in paper. For this part also any complicated index can be handled. In example below, j can be any arbitrary expression e.g. *a+b*c*.

int a[100];

```
1   int b[50];
2   int c[200];
3   c[j] = 0;
4     if(i > 0) {
5           b[j]=0;
6     } else {
7           j++;
8           a[j]=1;
9     }
10  }
```

i)Backward Modification:

```
# [entry]
        %1 = load i32, i32* %j, align 4, !dbg !41
        %idxprom1 = sext i32 %1 to i64, !dbg !42
        call void @__checkArrayBounds_max_i64_u64(i64 %idxprom1, i64 100)        → j<100
        call void @__checkArrayBounds_min_i64_u64(i64 %idxprom1, i64 0, i64 100) → j>0

# [if.then]          ;succs =    ;preds = entry,

        %7 = load i32, i32* %j, align 4, !dbg !55
        %idxprom6 = sext i32 %7 to i64, !dbg !56
        call void @__checkArrayBounds_max_i64_u64(i64 %idxprom6, i64 0, i64 50)  → j<50
        call void @__checkArrayBounds_min_i64_u64(i64 %idxprom1, i64 0, i64 50)  → j>0

# [if.else]          ;succs =    ;preds = entry,
        %11 = load i32, i32* %j, align 4, !dbg !65
        %idxprom12 = sext i32 %11 to i64, !dbg !66
        call void @__checkArrayBounds_max_i64_u64(i64 %idxprom1, i64 0, i64 100) → j<100
        call void @__checkArrayBounds_min_i64_u64(i64 %idxprom1, i64 0, i64 100) → j>0
```

ii)Forward Elimination:

```
# [entry]
        %1 = load i32, i32* %j, align 4, !dbg !41
        %idxprom1 = sext i32 %1 to i64, !dbg !42
        call void @__checkArrayBounds_max_i64_u64(i64 %idxprom1, i64 0, i64 100) → j<100
        call void @__checkArrayBounds_min_i64_u64(i64 %idxprom1, i64 0, i64 100) → j>0

# [if.then]          ;succs =    ;preds = entry,

        %7 = load i32, i32* %j, align 4, !dbg !55
        %idxprom6 = sext i32 %7 to i64, !dbg !56
        call void @__checkArrayBounds_max_i64_u64(i64 %idxprom6, i64 0, i64 50)  → j<50

# [if.else]          ;succs =    ;preds = entry,
```

```
%11 = load i32, i32* %j, align 4, !dbg !65
%idxprom12 = sext i32 %11 to i64, !dbg !66
```

Global elimination decreases size of STREAM executable to 25.8 kB.

C) Loop Optimizations

The purpose of loop optimizations is to reduce the number of bounds checks executed in loops. There are different kind of loop optimizations for bounds check removal. Here, I present the optimizations and a sample code for each type implemented in our project.

Known Loop Bounds Used for Access

Consider the code below:

```
1  int foo[10];
2  for (int i =0 ; i<10; i++)
3          foo[i] = …
```

The bounds checks in this case will be inserted before line 3. Therefore, the bounds will be in execution path of for loop. However, since the bounds of variable i is known, we can move this bound check outside the loop. Also, we just need to check Max and Min of the for loop with the bounds of array foo.

Partially Known Loop Bounds Used for Access (Monotonic)

Consider the code below:

```
1  int foo[10];
2  for (int i =0 ; ; i++) {
3          foo[i] = …
4          i++
5          if (foo[i])
6                  break;
7  }
```

In this case there is no known upper bound for the for loop. However, i is increasing monotonically. Therefore, we can move the Min check, like last cast, before the loop. However, we should keep the maximum check in the loop. The same method applies for minimum checks.

Known/Unknown Loop Bounds; But no Dependent Access (Invariant)

Consider the code below:

```
1  int foo[10];
2  int b = 4;
3  for (int i =0 ; i<10; i++) {
4          foo[f(b)] = …
5  }
```

In this code the access is not dependent on the loop variable. Also, there is no other assignment to b inside the loop. So, we can say b is invariant. Therefore, we can move the same bounds check outside the loop. Also we should duplicate all dependent instructions for checks and calculations outside the loop as well. (e.g., loads, adds). For this project, we moved all

dependent instruction before the loop. This will increase the register pressure, which is not covered in our project.

Known/Unknown Loop Bounds; Monotonic (not Dependent on Loop Variable)

Consider this code:

```
1  int foo[10];
2  int j = 1;
3  for (int i =0 ;i<10 ; i++) {
4          foo[j] = …
5          j+=2;
6  }
```

In this case we have access by a variable that is not the loop variable, however, it changes during the loop. In this cases, we find how the variable is changing, if it is monotonic, then we can move one of the bound checks outside the loop.

If/Else cases in the Loop

Consider the code below:

```
1  int foo[10];
2  for (int i =0 ; i<10; ) {
3          if () {
4                  foo[i] = …
5                  i+=2;
6          }
7          else
8                  foo[i++]=…
9  }
```

For this case, we have two checks for each one of the if and else statements. The bounds of two checks are different. We choose the weakest one and propagate it and do not touch the other one. This case can be much more complex that this example. For other complex cases we always prioritize safely to reduction of bounds checks.

Nested Loops (Mixed of Previous Cases)

Consider the code below:

```
1  int foo[10];
2  int boo[20];
3  int b = 2;
4  for (int i =0 ; i<10; )
5          for (int j =5 ; j<20; j++ ) {
6                  foo[i] = …
7                  foo[b] = …
8                  boo[j]=…
9  }
```

For this case, we might be able to move some checks to most-outer loop, while some checks can be moved just one loop above. Also, it is possible that we could not move checks at all (not shown in the example, for instance complex operations). For the example given we can move check accesses based on j one level up, and the check accesses based on i to the most-outer loop. Also, check accesses based on the constant b is moved the the most-outer loop.

• Improvements

1. **Constant propagation:** running constant propagation pass before our pass helps to reduce number of checks by improving number of identical indexes that we can find.
2. **Reassociating expressions:** using reassoc pass allong with constant propagation can increase efficiency of finding identical indexes. For example: $4 + (x + 5) \Rightarrow x + (4 + 5)$
3. **Global Value Numbering:** using gvn can also help to find more expressions with identical value.
4. **PRE:** Our implementation has PRE to some extend but more elaborate PRE can find identical indexes more aggressively.
5. **CFG modification**: changing CFG ,e.g. adding redundant basic blocks, as it is helpful in PRE sometimes, can be also helpful with better check redundancy elimination.