**Project Phase 2**

Jeremy Duvall, Chris Grayson, Jimmy O’Neill

# Naïve array bounds checking

Implementing a naïve bounds checking algorithm using the LLVM compiler infrastructure requires a simple algorithm that is aware of the method that LLVM uses to compute memory addresses. A basic module pass is set to iterate through all the functions in the input program, and augmented with a classic visitor pattern that specifically handles each GetElementPtrInst instruction (henceforth known as the GEP instruction). Inside the visitor implementation, a simple object model is built to handle indexing all inserts of a generated check function that handles the array bounds logic. The signature of the check function is simply the subscript requested along with the length of the array—both of which are available in-situ from the GEP instruction by walking it’s operand and operand type hierarchy. The bounds checking algorithm is quite fast, and it works for statically assigned arrays as well as with member fields on user defined structs. However, it does not handle aliasing, so indexing pointers that are assigned to an array will never be checked. The table below displays the wall clock running time for the insertion algorithm for each of the programs in a media suite provided by the TA.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Checks inserted** | **Wall time (s)** | **% decrease** |
| cjpeg.linked.rbc | 944 | 0.0150 |  |
| cjpeg.llvm.bc | 120 | 0.0051 | **686.67%** |
| encode.linked.rbc | 78 | 0.0006 |  |
| encode.llvm.bc | 9 | 0.0004 | **766.67%** |
| mpeg2decode.linked.rbc | 498 | 0.0034 |  |
| mpeg2decode.llvm.bc | 158 | 0.0022 | **215.19%** |
| toast.linked.rbc | 452 | 0.0032 |  |
| toast.llvm.bc | 303 | 0.0014 | **49.17%** |
| rawcaudio.linked.rbc | 6 | 0.0004 |  |
| rawcaudio.llvm.bc | 3 | 0.0003 | **100.00%** |
| rawdaudio.linked.rbc | 3 | 0.0005 |  |
| rawdaudio.llvm.bc | 6 | 0.0003 | **-50.00%** |

*Table 1*

In table 1 note that the first file in each duet has been compiled with PRE turned **off** while the second one has PRE turned **on**. As indicated by high percentage decreases in the number of checks inserted into the optimized program PRE clearly has a nice advantage for eliminating potential redundancy in array bounds indexing. The last program, however, exhibits an opposite trend. We suspect this could be due to code growth introduced by the PRE algorithm when hoisting certain sub-expressions to their earliest/latest blocks.

# Optimizing array bounds checking with ABCD

Certain array bounds checks are redundant, specifically those that occur when the program will provably never index an array out of bounds. To explore solutions to this problem, we implement a modified version of the Array Bounds Checks on Demand (ABCD) algorithm introduced by Gupta et al. In order to arrive at a complete solution, two issues need to be resolved. First, an extended SSA IR should be formulated to handle naming and processing of the constraints associated with the checks. Second, a constraint graph should be built from the values gleaned from the extended SSA IR. And finally, a solver must be constructed to ascertain if an array bounds check is clearly needed. This solver must be conservative in order to be correct—as removing an array bounds check that likely should exist is a bad idea. The following table illustrates how many constraints are removed from each of the programs referenced in Table 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **File** | **Checks inserted** | **Checks removed** | **Checks left** | | ***Wall time*** | **% removed** |
| cjpeg.linked.rbc | 944 | 608 | | 336 | *794.2167* | **64.41%** |
| cjpeg.llvm.bc | 120 | 62 | | 58 | *431.4052* | **51.67%** |
| encode.linked.rbc | 78 | 56 | | 22 | *16.2744* | **71.79%** |
| encode.llvm.bc | 9 | 6 | | 3 | *1.1543* | **66.67%** |
| mpeg2decode.linked.rbc | 498 | 368 | | 130 | *42.1838* | **73.90%** |
| mpeg2decode.llvm.bc | 158 | 63 | | 95 | *32.9546* | **39.87%** |
| toast.linked.rbc | 452 | 420 | | 32 | *89.5437* | **92.92%** |
| toast.llvm.bc | 303 | 230 | | 73 | *74.0850* | **75.91%** |
| rawcaudio.linked.rbc | 6 | 0 | | 6 | *0.0934* | **0.00%** |
| rawcaudio.llvm.bc | 3 | 0 | | 3 | *0.0437* | **0.00%** |
| rawdaudio.linked.rbc | 3 | 0 | | 3 | *0.1016* | **0.00%** |
| rawdaudio.llvm.bc | 6 | 0 | | 6 | *0.0305* | **0.00%** |

Below are implementation notes for the extended SSA and constraint graph + solver that have been implemented.

## Extended SSA

The class “eSSA” contains the data structures and routines that create an eSSA representation on top of the sparse representation that LLVM provides. Instead of directly modifying the provided SSA representation by inserting pi assignments and renaming variables, we use the std::map class to store three crucial mappings: a nested map from every instruction in the module to each name in the module to that SSA name’s corresponding eSSA name for that instruction (Instruction\*->SSA name->eSSA name); a nested map from every successive pair of basic blocks to the “eSSAedge” that contains the pi assignments that may exist between the blocks (BasicBlock\*->BasicBlock\*->eSSAedge\*); and a map from each check function call in the program to the corresponding “piAssignment” (CallInst\*->piAssignment\*). These maps are named eSSA::var\_map, eSSA::edges, and eSSA::check\_pi\_assignments, respectively. The extension of eSSA from SSA is performed in eSSA::SSA\_to\_eSSA(Module &m), and the renaming of the SSA variables is performed by eSSA::rename(DominatorTree &DT) for each function in the module. eSSA::rename makes use of the DominatorTree analysis pass provided by the LLVM framework to traverse the graph. The eSSA class is also responsible for traversing the module to detect constraints once the extension and renaming are complete. This is performed by eSSA::find\_constraints(), which provides the constraint graph construction classes with the relevant information to create the inequality graph and returns a vector of constraint graphs, one for each function in the module.

## Constraint Graph + Solver

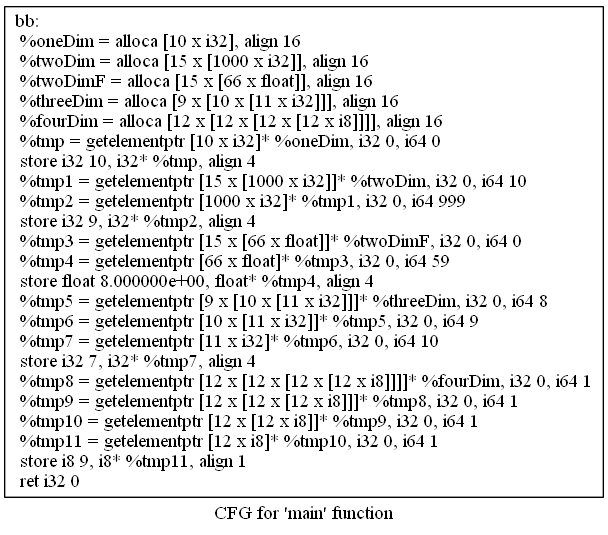
The constraint graph that we are using is constructed as per the instructions given in the ABCD paper. Our traversal algorithm, however, is slightly different. We do a normal depth first search from the graph node corresponding to the array length to the graph node corresponding to the array index. To handle loops, every time we add a node to a given traversal path we check to see if that node already exists in the path. If it does, we calculate the loop value. If the loop value is greater than zero, the traversal path is flagged to have a positive loop. Upon graph traversals arriving at the destination node, they are placed in to a vector containing all successful traversals. Once all traversals are completed, we iterate through them. If a single one of them has a positive loop or a single one has a traversal cost greater than -1, we say that the check is not redundant. Otherwise, the check is redundant and is thereby removed.

## Example CFG’s

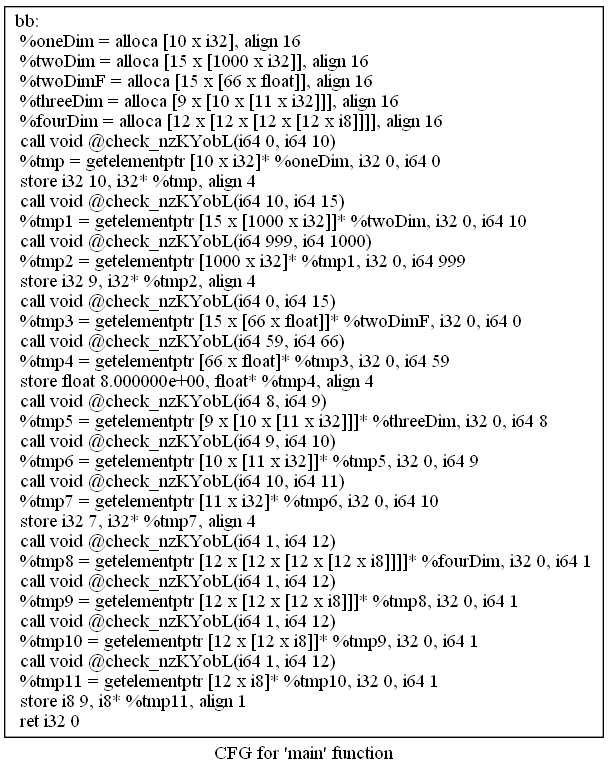
Below are some sample CFG’s illustrating our ABCD algorithm. In the first example, all checks are spurious and should be removed. In the second example, one check of a multi-dimensional array may go out of bounds, but the other is spurious. Example code is available in the tests directory to further illustrate this. Please see the README distributed with the code for more information.

## Example 1: tests/crazyarrays.c

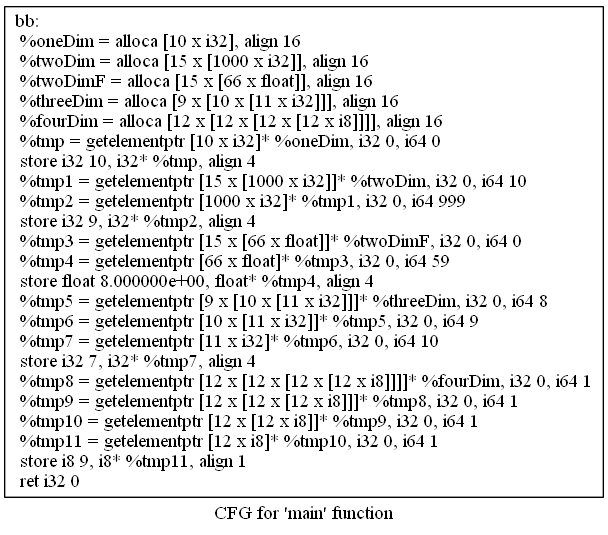
CFG before



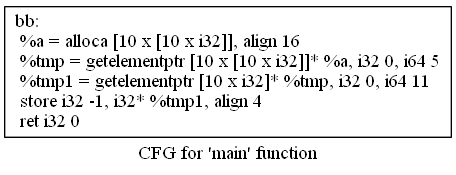
CFG after checks inserted



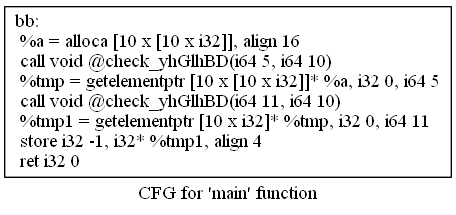
CFG after ABCD

  
Example 2: errloopmultidim.c

CFG before



CFG after checks inserted



CFG after ABCD

