Visualizing code structure in LLVM

Dmitry Mikushin

dmitry.mikushin@usi.ch

December 5, 2013

Abstract

This presentation overviews LLVM's capabilities for creating human-readable code structure visualizations and related dependence analysis passes.

Program code example (vector_sum)

```
void vector_sum(int length, float* in1, float* in2, float* out)
{
    for (int i = 0; i < length; i++)
        out[i] = in1[i] + in2[i];
}

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc != 2) return 1;
    int length = atoi(argv[1]);
    float* in1 = (float*)malloc(length * sizeof(float));
    float* in2 = (float*)malloc(length * sizeof(float));
    float* out = (float*)malloc(length * sizeof(float));
    for (int i = 0; i < length; i++)
    {
        in1[i] = drand48();
        in2[i] = drand48();
    }

    vector_sum(length, in1, in2, out);

    printf("%f\n", out[length - 1]);
    return 0;
}
```

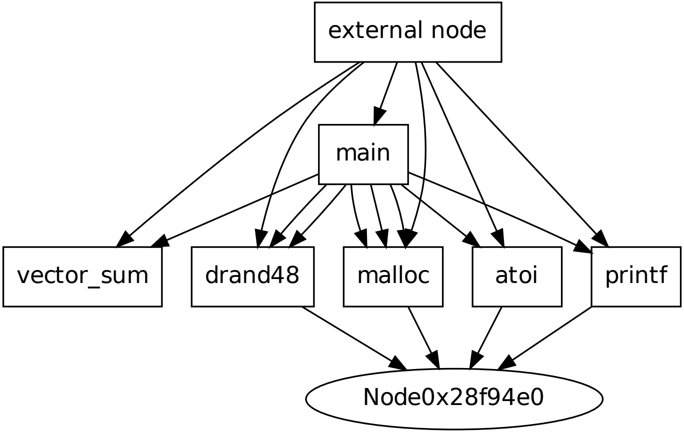
LLVM IR for the function of interest (vector_sum)

```
$ clang -emit-llvm example.c -O1 -S -o - | less
define void @vector_sum(i32 %length, float* nocapture readonly %in1, float* nocapture readonly %in2, float* nocapture %out) #0 {
entry:
  %cmp9 = icmp sgt i32 %length, 0
  br i1 %cmp9, label %for.body, label %for.end

for.body:                                         ; preds = %entry, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]
  %arrayidx = getelementptr inbounds float* %in1, i64 %indvars.iv
  %0 = load float* %arrayidx, align 4, !tbaa !1
  %arrayidx2 = getelementptr inbounds float* %in2, i64 %indvars.iv
  %1 = load float* %arrayidx2, align 4, !tbaa !1
  %add = fadd float %0, %1
  %arrayidx4 = getelementptr inbounds float* %out, i64 %indvars.iv
  store float %add, float* %arrayidx4, align 4, !tbaa !1
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body, %entry
  ret void
}
```

Callgraph, control & data flow

Two forms of graphs available in LLVM right from the command line:

1. **Call graph**
2. CFG (basic blocks)
3. CFG & DFG (instructions)

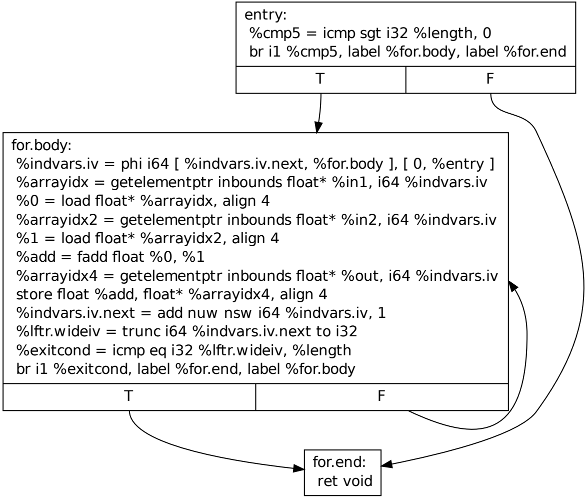


Call graph

Callgraph, control & data flow

Two forms of graphs available in LLVM right from the command line:

1. Call graph
2. **CFG (basic blocks)**
3. CFG & DFG (instructions)

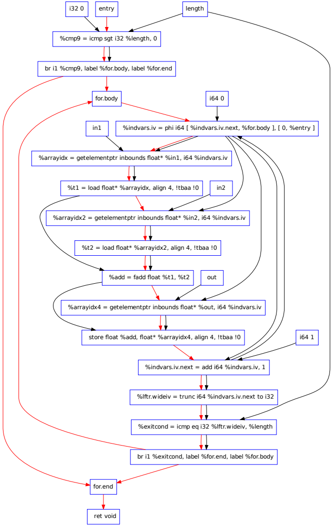


CFG for 'vector_sum' function

Callgraph, control & data flow

Two forms of graphs available in LLVM right from the command line:

1. Call graph
2. CFG (basic blocks)
3. **CFG & DFG (instructions)**



Black edges - dataflow, red edges - control flow

Callgraph, control & data flow

Spells:

1. Call graph – built-in LLVM pass + dot

```
$ clang -emit-llvm -S -c example.c -o - | opt -O1 | opt -dot-callgraph -o /dev/null && dot -Tpdf callgraph.dot -o callgraph.pdf
```

2. CFG (basic blocks) – built-in LLVM pass + dot

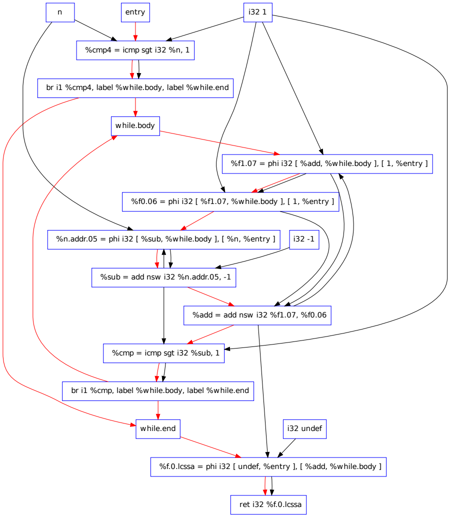
```
$ clang -emit-llvm -S -c example.c -o - | opt -O1 | opt -dot-cfg -o /dev/null && dot -Tpdf cfg.vector_sum.dot -o cfg.vector_sum.pdf
```

3. CFG & DFG (instructions) – llvmpy script by Paul Sokolovsky + dot
https://github.com/pfalcon/graph-llvm-ir

```
$ ./graph-llvm-ir ./example.ll && dot -Tpdf vector_sum.dot -o vector_sum.pdf
```

CFG & DFG of Zeller's Ex. 7.1

```
/* fibo.c -- Fibonacci C program to be debugged */
#include <stdio.h>
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1)
    {
        n = n — 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}
int main()
{
    int n = 9;
    while (n > 0)
    {
        printf("fib(%d)=%dN", n, fib(n));
        n = n — 1;
    }
    return 0;
}
```



Black edges - dataflow, red edges - control flow

Note uninitialized `int f` becomes `undef` →

Dependence Analysis passes

Two LLVM passes focused on dependencies:

1. Memory Dependence Analysis
2. Dependence Analysis

Both leverage Alias Analysis to reduce $O(n^2)$ strength.

Memory Dependence Analysis

- Fully functional
- Analyses dependencies between memory references:
  - **Clobber** – instruction that clobbers the memory, e.g. a may-aliased store
  - **Def** – instruction defines/produces the desired memory location
  - **NonLocal** – outside of the current basic block (need to check predecessors)
  - **NonFuncLocal** – outside of the current function
- No visual graph output
- Printed output is not very clear (see next slide)

Memory Dependence Analysis – Test

```
define void @vector_sum(i32 %length, float* nocapture readonly ↩
      %in1, float* nocapture readonly %in2, float* nocapture ↩
      %out) #0 {
entry:
  %cmp9 = icmp sgt i32 %length, 0
  br i1 %cmp9, label %for.body, label %for.end

for.body:                              ; preds = %entry, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, ↩
        %entry ]
  %arrayidx = getelementptr inbounds float* %in1, i64 ↩
        %indvars.iv
  %0 = load float* %arrayidx, align 4, !tbaa !1
  %arrayidx2 = getelementptr inbounds float* %in2, i64 ↩
        %indvars.iv
  %1 = load float* %arrayidx2, align 4, !tbaa !1
  %add = fadd float %0, %1
  %arrayidx4 = getelementptr inbounds float* %out, i64 ↩
        %indvars.iv
  store float %add, float* %arrayidx4, align 4, !tbaa !1
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                               ; preds = %for.body, %entry
  ret void
}
```

Memory Dependency Analysis printed output:

```
$ clang -emit-llvm example.c -O1 -S -o - | opt -memdep -print-↩
      memdeps -o /dev/null
Function vector_sum memory dependencies:

    Unknown in block %for.body
    Unknown in block %entry
  %0 = load float* %arrayidx, align 4, !tbaa !1

    Unknown in block %for.body
    Unknown in block %entry
  %1 = load float* %arrayidx2, align 4, !tbaa !1

    Def from:   %1 = load float* %arrayidx2, align 4, !tbaa !1
  store float %add, float* %arrayidx4, align 4, !tbaa !1
```

Why `store` depends only on one `load` out of two?..

Dependence Analysis

- Work in progress
- Implemented by Preston Briggs following
  “Practical Dependence Testing” by Goff, Kennedy, Tseng – PLDI 1991
- Comprehensive dependence analysis:
  - **Output** – write after write
  - **Flow (true)** – read after write
  - **Anti** – write after read
  - ...
- No visual graph output

Conclusion

So far:

- General dependence analysis is work in progress in LLVM
- No graph visualizations based on dependence analysis are implemented