LLVM Project 2 Report

Henry Yichao Dong

Introduction

In this project, a wide range of LLVM transformation passes on the C language were explored, and the following aspects of the passes have been looked at specifically:

1. The nature of the transformation done by each pass.
2. The efficacy of the resultant optimization measured by the number of LLVM IR instructions as well as code execution wall time.
3. The effect of using a combination of different passes and experiments regarding to the optimal sequence of the passes to apply to code optimization in general, and also the different behaviors of particular transformation sequences on specific kernel codes.

Method

1. The benchmark source code

To benchmark an LLVM pass, I typically use the following set-up:

```c
#include <stdio.h>
#include <sys/time.h>

int main()
{
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);
    long int n = 10000000000;
    for (long int i = 0; i < n; i++) {
        //optimizable code
    }
    gettimeofday(&end_time, NULL);
    time_t elasped_time_usec = (end_time.tv_sec*1e6 + end_time.tv_usec)
            - (start_time.tv_sec *1e6 + start_time.tv_usec);
    printf("Run time: %.4g seconds\n", elasped_time_usec*1e-6);
    return 0;
}
```

The code to be optimized is placed inside a loop and is executed *n* times to amplify the benchmark, and the execution time is measured by the UNIX system function *gettimeofday()* with microseconds resolution and the program execution time is reported in seconds accurate to 4 significant figures. (Please note that this benchmark code only works on UNIX systems including Linux and OSX).

2. Procedures to apply the LLVM optimization pass and benchmark study

To apply an LLVM transformation pass, we first compile the source code into byte code (.bc), onto which we can run the analysis pass "*instruction count*" to count the number of different types of instructions, and the specific transformation pass in question. For example, to test the *-mem2reg* transformation pass, I use the following sequence of commands in the bash console:

```
1 clang -emit-llvm -O0 -c test.c -o test.bc
2 opt  -mem2reg test.bc -o mem2reg.bc #applies the transformation
3 llvm-dis test.bc # makes readable test.ll
4 llvm-dis mem2reg.bc # make readable mem2reg.ll
5 opt -stats -analyze -instcount test.bc -S # counts number of instr
6 opt -stats -analyze -instcount mem2reg.bc -S #counts number of instr
7 lli test.bc #run unoptimized benchmark code
8 lli mem2reg.bc #run optimized benchmark code
```

This allows the user to compare the performance of the LLVM transformation by:

1. Compare the readable IR codes (the *.ll* files) to see what transformations have been made.
2. Compare the number and types of instructions generated before and after the transformation via the *-instcount* pass (line 5 and line 6 in above)
3. Execute both versions of the benchmark code to compare execution wall time (line 7 and line 8 in above). This step is repeated five times and we report the average program execution time for the benchmark.

Results

In this project, I looked at *6 different aspects* of LLVM transformations (in total 11 LLVM transformation passes):

1. Memory related optimizations:

Promoting memory to register (-mem2reg)
Memcpy optimization (-memcpyopt)

2. Dead code elimination optimizations:

Dead Code Elimination (-dce)
Dead Instruction Elimination (-die)
Dead Store Elimination (-dse)
Dead Global Elimination (-globaldce): removes dead internal globals
Dead Argument Elimination (-deadargelim)

3. Constant propagation optimizations:

Simple Constant Propagation (-constprop)

4. Reassociate expressions optimizations:

Reassociate Expressions (-reassociate)

5. Transformation of induction variables and loop optimizations:

Loop Strength Reduction (-loop-reduce)
Loop Invariant Code Motion (-licm)

6. Function Inlining:

Function Inlining (-inline)

In this section, we will first look at the effect of running each transformation passes individually, then we will continue on to test the best sequence of applying these passes to some sample codes.

1. Memory related optimizations:

Promoting memory to register (*-mem2reg*)

This optimization, the *-mem2reg*, is arguably one of the most important transformation passes in LLVM. This transformation promotes memory references to be register references, and in this process, it constructs the SSA form of the code and promotes the *alloca* instructions to specific registers on the machine. Because accessing register-stored variables is many magnitudes faster than accessing the variables allocated on the stack, this optimization is expected to yield a significant improvement in execution time

```
int add (int a, int b){
    return a + b;
}

int main(){
    ...
    for (long int i = 0; i < n; i++) {
        int a = 10, b = 10;
        if (i%2){
            add (a, b);
        }
    }
    ...
    return 0;
}
```

Fig 1. The benchmark code for *-mem2reg* (Here the … part is the same as the benchmark sample code mentioned above, and $n = 10^{10}$)

	Original	Optimized	%Change
Num of Alloca Inst	10	2	-80%
Num of Load Inst	13	4	-69%
Num of Memory Inst	40	14	-65%
Total Num of Inst	65	40	-38%
Exec Time (sec)	26.33	18.95	-28%

Table 1. The benchmark for *-mem2reg*

Clearly as expected, there is a large reduction on memory-related instructions (such as *alloca* and *load* instructions) after applying the transformation, and the resultant speed up is also quite significant for this very simple kernel code (a 28% reduction). For a more complicated source code with a large amount of variables, I believe the speed-up could be even greater.
In addition, the *-mem2reg* transformation is important because this transformation is needed for many other transformation to work on the IR, this is because many transformations in LLVM assumes the user has run the *-mem2reg* at least once before applying that particular transformations. One example of *-mem2reg* -dependant transformation is *-constprop* (constant propagation), as we will discuss later.

Memcpy optimization (-memcpyopt)

This optimization is supposed to perform various transformations related to eliminating memcpy calls, or transforming sets of stores into memsets. However I tried to apply this transformation on many different sample codes, and it did not have an effect on transforming the IR code. I suspect that this particular transformation has to work in conjunction with some other transformations, because it works particularly on the LLVM-specific *memcpy* and *memset* instructions.

2. Dead code elimination optimizations:

This class of optimizations includes passes such as
Dead Code Elimination (*-dce*): similar to *-die*, but rechecks newly dead instructions
Dead Instruction Elimination (*-die*): in a single pass eliminates obviously dead codes
Dead Store Elimination (*-dse*): removes redundant store instructions
Dead Global Elimination *(-globaldce*): removes dead internal globals
Dead Argument Elimination (*-deadargelim*): removes dead function arguments

Because the *-dce* and *-die* transformations are essentially the same, so here we only investigate the effect of *-dce,* which is supposed to be smarter than *-die*

```
 4 int g_a =10; //dead global
 5
 6 int use_var(int var, int dead_arg){
 7     dead_arg*= 13168%17; //dead code
 8     return var + 1;
 9 }
10
11 int main(){
12     ...
13     int a, b;
14     for (long int i = 0; i < n; i++) {
15         b = i*13168%17; //dead code
16         b = 10;
17         a = use_var(1, 0); //dead code
18         a = 10;
19     }
20     use_var(a, b);//b is a dead arg
21     ...
22     return 0;
23 }
```

Fig 2. The benchmark code for the dead code elimination transformations ($n = 10^{10}$)

Surprisingly, amongst the 5 different passes, only the dead store elimination (*-dse*) has changed the IR code and hence optimized the code, but only to a small extent (it only eliminates the dead variable inside function *use_var*). All other 4 optimizations have no effect at all. A search for documentation has revealed that the pass *-mem2reg* is required to run before for many other LLVM passes in order for the optimizations to work, and this is indeed the case.

Running *-mem2reg* in conjunction with the dead code elimination passes effectively eliminates the dead code, for example:
-dse: eliminates the unused variable *dead_arg* at line 7 in *use_var ()*
-dce: same as -dse, but in addition it eliminates every thing in the loop except line 17
-deadargelim: it changes the unused argument *dead_arg* to *undefined* in *use_var()*
The transformations *-globaldce* still has no effect.
For all optimizations above, the unreachable else branch is not eliminated. And the *for* loop (which is an empty loop) effectively, is not eliminated also.

Through multiple tests, the optimal sequence of transformation to use for this kernel code is -dce *-mem2reg -deadargelim -dse,* and this is compared to performing *-mem2reg* alone:

	-mem2reg alone	-mem2reg + dce passes	%Change
Total Num of Instr	41	37	-9.8%
Execution Time (sec)	27.81	20.15	-28%

Table 2. The benchmark for *dead code elimination passes*

Note there is also an more aggressive version (*-adce*) of dead code elimination, but replacing the *-dce* with this more aggressive pass seems to have no effect on this sample code.
The reduced instructions corresponds to the eliminated dead code described above, while the improvement in execution time is highly dependent on the amount and nature of the dead code being eliminated. In my sample code, the dead codes are computationally intensive and are executed many times in the loop, so here I have 28% improvement. But still, note that these dead code elimination schemes are quite basic: the dead code at line 17 has not been eliminated, and the same for the empty for loop and the unreachable else branch.

3. Constant propagation optimizations:

Here I only looked at the pass *Simple Constant Propagation* (-constprop), which as its name suggests, only looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction.

```
12 int use_var(int a, int b){
13     return a + b;
14 }
15
16 int main(){
17     ...
18     int a, b;
19     for (long int i = 0; i < n; i++) {
20         a = 10;
21         b = 10;
22         a = a + b; //should be propagated as a = 20
23         b = b + 20; //should be propagated as b = 30
24     }
25     use_var(a, b);//should be propagated as use_var(20, 30)
26     ...
27     return 0;
28 }
```

Fig 3. The benchmark code for simple constant propagations ($n = 10^{10}$)

As discussed earlier, the constant propagation pass is one of the passes that depends on the *-mem2reg* pass, which means by running the *-constprop* pass alone there is no transformation done in the IR code. In conjunction with *-mem2reg,* the constant propagation pass successfully propagated the constant values held in variables *a* and *b* all the way to line 25 (so *use_var(20, 30)* is called*)*, making lines 20 - 23 dead codes to be removed by the *-mem2reg* transform.

	-mem2reg alone	-mem2reg + -constprop	%Change
Total Num of Instr	40	36	-10%
Execution Time (sec)	3.378	3.321	-1.7%

Table 3. The benchmark for *dead code elimination* passes

Here we can see this transformation only makes a small improvement on execution time, because we only eliminated a few (not so costly) instructions.

4. Reassociate expressions optimizations:

This transformation simplifies computation by re-arranging the order of commutative operations in a way that promotes better constant propagation, LICM and etc.

```
 5 int use_var(int a, int b){
 6     return a + b;
 7 }
 8
 9 int main(){
10     ...
11     int a = 10, b = 10;
12     for (long int i = 0; i < n; i++) {
13         // equiv of a = 5 + 5 - 10 + b - b = 0
14         a = (5 + b) + 5 - (b - 10);
15         // equiv of b = 5 + 5 - 10 + a - a = 0
16         b = (5 + a) - 5 - (a - 10);
17     }
18     use_var (a, b);
19     ...
20     return 0;
21 }
```

Fig 4. The benchmark code for *reassociate expressions* ($n = 10^{10}$)

This pass works by itself or with *-mem2reg*, but for comparison purpose here I run it in conjunction with *-mem2reg*.

	-mem2reg alone	-mem2reg + -constprop	%Change
Total Num of Instr	46	42	-9.7%
Execution Time (sec)	14.39	3.385	-76.5%

Table 4. The benchmark for the *reassociation* pass

Surprisingly, this pass produced more than 3-fold speed up over the unoptimized code. By looking at the transformed IR, in additional to the reassociation, which definitely reduces a few arithmetic computations for each iteration in the loop, but more

prominently, in the unoptimized version, the values of *a* and *b* are represented as *phi nodes*, and each time at the branch condition, the values of these two variables has to be evaluated by the lengthy computations within the loop; yet this work is waived in the optimized code.

5. Transformation of induction variables and loop optimizations:

Optimizing loops is critical for code performance, and the common strategies are to simply computations involving the induction variables and hoisting the loop invariants out of the loop scope. Here I looked at 3 different passes:
On the other hand, the Loop Strength Reduction (*-loop-reduce*) pass performs a strength reduction on variables inside loops that have as one or more of their components the loop induction variable.
The Loop Invariant Code Motion (*-licm*) pass attempts to remove invariant variables out of the loops.
There is also Canonicalize Induction Variables (*-indvars*) transformation pass in the LLVM, but according to the current online documents, this transformation does not canonicalize the loops any more, and many other loop optimizations can transform the code better without running this pass. In practice, I found that including this optimization along with the two loop optimizations discussed above actually increases the program execution time.

```
11 int use_var (int a, int b){
12     return a+b;
13 }
14
15 int main(){
16     ...
17     long int n = 100000;
18     int invariant, a, b;
19     int arr[100][100];
20
21     for (long int i = 0; i*i < n*n; i++) {//Induction variable simplification
22         for (int j = 0; j*j < 100*100; j++){//Induction variable simplification
23             for (int k = 0; k*k <100*100; k++){ //Induction variable simplification
24                 invariant = n%17; //LICM
25                 a = j*4 + invariant;//Loop Strength Reduction
26                 b = k*8;//Loop Strength Reduction
27                 arr[j][k] =use_var(a, b);
28             }
29         }
30     }
31     ...
32     return 0;
33 }
```

Fig 5. The benchmark code for loop optimizations ($n = 10^5$)

For benchmarking, I compared two versions: one with only *-mem2reg* (the original) and one with *-mem2reg -loop-reduce -licm.*
Looking at the optimized IR code, the Loop Strength Reduction (*-loop-reduce)* optimization simplies the loop conditions by canonicalize the loop induction variables, *i, j,* and *k* in this example.
Applying the LICM also hoist the loop invariant out of the loop scope. As a result, we have increased performance:

	Original	Loop Optimized	%Change
Total Num of Instr	63	61	-3.2%
Execution Time (sec)	3.385	2.238	-33.9%

Table 5. The benchmark for the *loop optimization* passes

Note although the %Change in total number of instructions is that large, the code transformation inside the loop body is significant: there are far less instructions inside each loop body and more instructions outside the loop. Hence the total number of executed instructions is reduced and hence we have a sizable speed improvement.

6. Function Inlining

Function inlining is an important optimization in LLVM. For short functions especially, inlining it into the caller code effectively reduces the function call-stack overhead, and hence can predictably reduce program run time in this scenario.

```
5  int func (int a, int b){
6      a++, b--;
7      return a + b;
8  }
9
10 int main(){
11     ...
12     int a = 10, b = 10, temp;
13     for (long int i = 0; i < n; i++) {
14         temp = func (a, b);//inlinable function
15     }
16     ...
17     return 0;
18 }
```

Fig 6. The benchmark code for function inlining ($n = 10^{10}$)

Again, this pass works by itself or with *-mem2reg*, but it performs much better if we run *-mem2reg* first. For comparison purpose here I run it in conjunction with *-mem2reg*.

	Original	Inline Optimized	%Change
Total Num of Instr	38	37	-2.6%
Execution Time (sec)	19.40	3.73	-80.8%

Table 6. The benchmark for the *function inlining* pass

As evidence from the benchmark, function inlining is really useful if the same short function is being called many times in a loop. For this case, by looking at the IR, I was surprised to see that this pass has optimized the arithmetic computation within the inlined func() as well, by propagating the constants and results in any empty loop body (it did not remove the loop entirely, though). Therefore, this inline pass does more work than its name suggests.

Experiment with the best sequence to apply the transformations

Studying the combined effect of applying multiple transformations rather than focusing on a single-category of transformations like in above is inherently more difficult, as there are dependencies in the code optimization process between certain optimizations.

As a general strategy, I hypothesize that there is an optimal sequence of transformations that we can perform. For example, *constant propagation* operations often makes certain instructions dead, and therefore it would make sense to apply *dead code elimination* passes afterwards. And since many LLVM optimization passes depends on *-mem2reg*, it is also a common practice to run this transformation before others.

For this particular experiment, I wrote a piece of kernel code that has various opportunities for optimizations, and tests were carried out to figure out the best sequence.

```c
#define ARRAY_SIZE 1000
struct foo {
    int m;
    int arr[ARRAY_SIZE];
};

int use_var (int a, int b){
    return a+b;
}

int reassociate (int num1, int num2, int dead_arg){
   return (100 + num1) -50 + ((num2 - 50) + 50) - 50; // Equal to num1 + num2
}

int main(){
    ...
    long int n = 10000;
    int a = 1, b = 1, temp;
    int dead_arr[100][100];

    for (long int i = 0; i < n; i++) {
        struct foo f; //dead code
        for (int j = 0; j*j < 100*100; j++){//Loop Strength Reduction
            b = 131683%29*1613895%129*76; //Dead code
            for (int k = 0; k*k <100*100; k++){ //Loop Strength Reduction
                b = 10; //Constant propagation makes this dead code
                a = 131683%29*1613895%129*76; // a loop invariant and is dead here
                a = b + 10;
                b = reassociate(a, b, f.m);
                a = j*4 + b;//Loop Strength Reduction
                b = k*8;//Loop Strength Reduction
                temp =use_var(a, b);
                dead_arr[j][k] = a + b; //dead code
            }
        }
    }
    ...
    return 0;
}
```

Fig 7a. The benchmark code for combined transformation passes

With this piece of sample code, I started testing with the combined transformation passes from all 6 categories that are discussed in above.
After testing ALL possible sequences, the following sequences is one of the many sequences that gives the best speed-up, at around 80% reduction at execution time (see Table 7a):
opt -mem2reg -inline -loop-simplify -reassociate -constprop -sccp -deadargelim -dce -indvars -loop-reduce -licm -adce test.bc -o all.bc

But are all these passes necessary? Can we do even better than this?
After some careful selection, I came up with the following minimal sequence of passes that maintains the optimization performance (see Table 7a):
opt -mem2reg -inline -reassociate -indvars -loop-reduce test.bc -o all.bc

Surprisingly, this time we used only 5 passes out of the full 12 passes in above, and the bytecode this shorter sequence generated is ***identical*** to the longer sequence of passes. Therefore certain LLVM passes are redundant (at least in this example code).

The effect of the order of application of the difference passes is also briefly investigated:
1. It turns out for this sample piece of code, the order we perform *-mem2reg* and *-inline* is most important, it is best to apply these two transformations as the first two transformations.
2. Order of applying other groups of passes does not seem to have a large impact on the optimization performance in my sample code. And this is counter-intuitive, because one can expect that there must be some dependence between the passes. For example, constant propagation and reassociation often leads to dead codes, and as a result, dead code elimination should be applied after these two transformations. Yet the order does not seem to matter in this piece of kernel code.

	Original	Optimized	%Change
Total Num of Instr	76	69	-2.6%
Execution Time (sec)	47.99	6.05	-80.8%

Table 7a. The benchmark for combined sequence of passes

I was also intrigued to see how this simple LLVM optimization scheme applies to more complicated scenario. The following is a slightly more sophisticated kernel code.

```c
#define ARRAY_SIZE 1000
struct foo {
    int m;
    int arr[ARRAY_SIZE];
};

void init_foo (struct foo* f){
    f->m = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        f->arr[i] = 1;
    }
}

int use_var (int a, int b){
    return a+b;
}

/*To test if it can be optimized to the more efficient pass-by-pointer*/
int use_foo (const struct foo f){
    int sum;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += f.arr[i];
        sum %= 100;
    }
    return sum;
}

int reassociate (int num1, int num2, int dead_arg){
   return (100 + num1) -50 + (num2 - 50); // Equal to num1 + num2
}

int main(){
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);
    long int n = 3000;
    int a = 1, b = 1, ret1, ret2;
    int dead_arr[100][100];

    for (long int i = 0; i < n; i++) {
        struct foo f;
        for (int j = 0; j*j < 100*100; j++){//Loop Strength Reduction
            b = 131683%29*1613895%129*76; //Dead code
            for (int k = 0; k*k <100*100; k++){ //Loop Strength Reduction
                b = 10; //Constant propagation makes this dead code
                struct foo dead_f;
                init_foo(&dead_f);//dead_f should be eliminated by dce
                a = 131683%29*1613895%129*76; // a loop invariant
                a = b + 10;
                init_foo(&f);
                f.m = reassociate(a, b, f.m);
                a = j*4 + b;//Loop Strength Reduction
                b = k*8;//Loop Strength Reduction
                ret1 =use_var(a, b);
                dead_arr[j][k] = a + b; //dead code
            }
        }
        ret2 = use_foo(f); //Only the very last temp in the loop is not dead
    }
    ...
    return 0;
}
```

Fig 7b. A more complicated benchmark code for combined transformation passes

Very surprisingly, this time running the previously optimal (either longer or shorter) sequence of transformation pass on this piece of code both *increases* the number of instructions and also has virtually *no improvement* on code execution time.

	Original	Optimized Long	%Change
Total Num of Instr	110	144	-2.6%
Execution Time (sec)	21.4	21.1	-1.4%
	Original	**Optimized Short**	**%Change**
Total Num of Instr	110	147	34%
Execution Time (sec)	21.1	20.8	-1.4%

Table 7b. The benchmark for combined sequence of passes on the more complicated code

So this time, those passes that worked very in the simpler optimization case no longer worked at this more complicated code. The introduction of a *struct* and a function that uses *struct* (*or struct**), poses a challenge to this optimization routine. This behavior is interesting I hypothesize that LLVM needs additional passes to deal with complicated types such as a structure.

Conclusion

This study looked at six different aspects of LLVM transformation passes, which admittedly, is only a very small set of optimizations within the LLVM framework. Nevertheless, through some simple experiments and benchmark studies, one can appreciate how effective the LLVM transformation passes are in optimizing the code performance.

However, it is necessary to address some of the limitations on the methodologies that I used in this study:

1. To make the benchmarking more accurate, it is better to use a profiler that takes into the account of the stochastic behavior of the machine itself and performs better statistical treatment to the benchmark.

2. The sample source code is quite simple and certain naïve, and they are largely deterministic. It is much intriguing if I can test the passes on a larger scale, and more complicated source code.

3. It will be interesting to look at transformations involving Control Flow Graph in the future. Almost all of my test code is done without a *branching "if"* statement, which is certainly not realistic in the real world.

We can also see that the particular sequence of applying the optimizations can be important, and also the effect of optimization is highly code-dependent. To achieve better, and more general optimization, one really needs to have a deep understanding of the

LLVM framework and utilize many of the transformation passes in the *correct order*. For example, the common *-O1* optimization done by LLVM typically uses more than 40 different passes. This also gives us a glimpse of how complicated the LLVM framework is.

References

1. LLVM's Analysis and Transform Passes
http://llvm.org/docs/Passes.html#passes-licm

2. The stackoverflow page that discusses the canonical optimizations done by LLVM:
http://stackoverflow.com/questions/15548023/clang-optimization-levels