A High Level Domain-Specific Language For Compiler Escalation

Aditya Patel, Yeshwant Saini, Utkarsh Dwivedi

Abstract: The coding of compiler advancements alongside the program investigation required to do as such is a strenuous undertaking including a huge number of lines of code. This paper presents Co-Ex, an abnormal state area explicit language (DSL) for the utilization of compiler engineers as a device to be utilized in the compiler advancement pipeline for the age of compiler streaming agents. A basic, instinctive and succinct way for indicating compiler improvements is displayed by the DSL. The paper incorporates a model usage of the DSL which supports a lot of ten enhancements including nearby, worldwide and circle improvements. On accumulation of a Co-Ex program, the Co-Ex compiler makes an interpretation of the Co-Ex program to a LLVM C++ pass. This created pass goes about as the code streaming agent which takes a shot at LLVM transitional portrayal (LLVM IR) to upgrade it. Co-Ex has been created after cautiously examining low-level compiler framework, to be specific LLVM, just as the calculations for worldwide, neighborhood and circle compiler enhancements and this has been disconnected into suitable builds in the DSL. While apparatuses, for example, Lex and Yacc aid the age of the frontend periods of the compiler, to be specific the lexer and the parser, Co-Ex aids the age of compiler enhancers. This will give compiler designers a simple strategy for determining compiler enhancements and spares them the problem of perusing, getting, coding and troubleshooting low-level foundation code via naturally producing compiler streaming agents.

Index Terms: domain specific language, Co-Ex, compiler escalation generation, LLVM IR)

1. INTRODUCTION

The term compiler was coined by Grace Hopper, who expounds her compiler, the A-0 compiler [1], as "(The programmer) needs only to be able to use the catalogue to supply information to the computer about his problem. The UNIVAC, on the basis of the information supplied by the mathematician, under the control of a compiling routine, using subroutines and its own instruction code, produces a program. This program, in turn directs the UNIVAC through the computation on the input data and the desired results are produced.” Through the years, this remains to be the fundamental task of a compiler. However, with the prominence of compilers in computing and technology, there is an increasing need to make the output of compilers efficient. The basic task of a compiler involves converting a given program from source language to some target language. These source and target languages differ based on the application the compiler is designed for. A commonality between these applications is that the target language produced needs to be made as efficient as possible without slowing down the process of compilation too much. To do so, a deliberated combination of code optimizations needs to be performed. Several tools exist to assist in the generation of the initial phases of the compiler such as lexical analysis and parsing. Lex [2] and Yacc [3], Flex [4] and Bison [5], PLY [6] are some of them. The aim of this work is to present a tool called Co-Ex for compiler developers in the form of a high level language in which programs can be written, which when compiled result in the generation of compiler escalations. This provides the means for compiler developers to easily describe different compiler optimizations and eliminates the need for compiler developers to have to learn the intricacies of different low level compiler infrastructures - which are specific to different intermediate representations - and the need to write complex, lengthy compiler optimization code. The optimizations that can be specified in Co-Ex include global, local and loop optimizations. Co-Ex programs are then compiled into LLVM C++ passes. LLVM is a framework that is used in the development of compiler technologies. LLVM also provides the intermediate code/representation (IR) to be optimized called LLVM IR which is in static single assignment form. It has libraries that are used for the different optimization programs which transform and optimize the IR. The core libraries present in LLVM provide support for code optimization and code generation based on LLVM IR. The most attractive feature about LLVM IR is that it is independent of the source and target. The contributions of this work are threefold:

- A set of ten modules each of which perform one particular compiler optimization have been written in LLVM C++. Libraries in the LLVM framework as well as several routines from the LLVM open source code have been extensively used in the development of these modules which run on LLVM IR and optimize it.
- The language, Co-Ex, has been designed to concisely and accurately represent compiler optimizations.
- A compiler for Co-Ex has been developed which translates Co-Ex programs to C++ LLVM passes.

Aditya Patel, Assistant Professor, Department of CSE, LNCT College, Bhopal, India, Email: adityapatemail@gmail.com
Utkarsh Dwivedi, Student, Department of CSE, LNCT College, Bhopal, India, Email: utkarshdwivedibtech@gmail.com
Yeshwant Saini, Student, Department of CSE, LNCT College, Bhopal, India, Email: yashwantsaini1998@gmail.com

Fig. 1. Block Diagram Representing the Co-Ex System
2. RELATED WORK

Existing work has addressed the development of domain specific languages to assist in specifying, performing and verifying peephole optimizations. Alive [7], for example, is a domain specific language for formal verification of peephole optimizations, which runs on LLVM IR. The language was designed so that it is similar to the LLVM transformation that is being described, both syntactically and semantically. It is automatically translated to LLVM C++. Alive, however, is restricted to peephole optimizations. Automatic generation of compiler optimizations which is done based on example programs has been carried out by Tate et al. [8]. This approach compares these example programs before and after they are made efficient and tries to generalize optimization instances to optimization rules. OPTRAN [9] is a domain specific language which takes in specifications in the form of attributed abstract syntax trees and generates Pascal programs. In the work by Willcock et al. [10], the specification of optimizations is in the form of generic interface descriptions. The CoSy model [11] provides to compiler developers a mechanism to embed in optimizing transformations through engines that are provided a logical view on intermediate representations and interact with each other. Another tool that aids in the generation of escalations is OPTIMIX [12], which is based on graph rewriting systems. OPTGEN [13] is a tool that generates local optimizations given a set of operations and their cost along with a cost limit. All possible local optimizations within this cost limit are generated. The language presented in this paper, Co-Ex, allows compiler writers to specify a set of both local and global optimizations as well as loop optimizations. We provide a DSL for a small set of optimizations and the dataflow analyses required to perform these optimizations. The specification of optimizations in Co-Ex is intuitive and compiler developers are given the freedom to specify their preferred nuances in the algorithms of the optimizations. A variation of this work has been done in which compiler optimizations are specified with the use of dependence relations and calculus [14]. This work, however, is targeted at the Stanford University Intermediate Format (SUIF) only. Meanwhile, the compiler that has been developed for Co-Ex runs on LLVM IR, which is language and target independent. Moreover, Co-Ex programs could be extended to be compiled to run on other intermediate formats as well.

3. THE CO-EX LANGUAGE

The Co-Ex language prototype has been designed to allow specifications for a set of ten optimizations. The list of optimizations allowed in Co-Ex presently includes:

- loop invariant code motion
- loop unrolling
- global value numbering
- tail recursion elimination
- induction variable simplification
- dead code elimination
- global common subexpression elimination
- local common subexpression elimination
- function inlining
- constant folding and propagation

The above optimizations have been written in LLVM C++ using existing libraries and available open source code. A sample input of LLVM IR is seen in Listing 1. Listing 2 and listing 3 illustrate the resulting IR after optimization has been performed on it. Listing 2 shows the IR after global value numbering has been performed while Listing 3 shows the result after the dead code elimination optimization has been carried out on the IR depicted in Listing 2.

Listing 1. LLVM IR Input Before Optimization
```
define double @test1(double %x, double %y)
  %add1 = fadd double %x, %y, !fpmath !0
  %add2 = fadd double %x, %y
  %foo = fadd double %add1, %add2
  ret double %foo
```

Listing 2. LLVM IR After Global Value Numbering
```
define double @test1(double %x, double %y)
  %add1 = fadd double %x, %y, !fpmath !0
  %add2 = fadd double %x, %y
  %foo = fadd double %add1, %add1
  ret double %foo
```

Listing 3. LLVM IR After Dead Code Elimination
```
define double @test1(double %x, double %y)
  %add1 = fadd double %x, %y, !fpmath !0
  %foo = fadd double %add1, %add1
  ret double %foo
```

a. Design of Co-Ex

Co-Ex provides flexibility to compiler developers to specify the algorithms of different optimizations. It aims to provide ease of use and sufficient utility to the compiler designers. The language has been designed by taking into consideration the traditionally followed algorithms for performing compiler optimizations along with the dataflow analyses required to perform them. Writing hardware-independent compiler optimizations with the help of a high-level DSL such as Co-Ex would allow developers to side-step the low-level implementation details of compiler infrastructures such as LLVM and GCC and specify optimizations in a compiler-independent manner.

b. Structure of a Co-Ex Program

1) opt: Every optimization needs to be declared within its own block along with the name of the optimization as shown in Listing 4. Listing 5 shows the respective translation. Along with this, appropriate header files are generated based on the optimization. The optimization names allowed are FunctionInlining, ConstantFolding, DeadCodeElimination, LocalCSE, IndVarSimplify, GlobalIVN, LoopInvariantCodeMotion, TailCallElim, LoopUnrolling and GlobalCSE.

Listing 4. Outermost Mandatory Optimization Block
```
opt : Optimization
{
  runOn(iterable);
}
```

Listing 5. Translation of Code in Listing 4
```
class OptimizationName : public FunctionPass
```
The above line generates files that perform the analysis and these files are included as header files in the main optimization program. reachingdefs can then be passed as a parameter to functions that require reaching definitions information to be computed.

Available Expressions
availableexprs:
findAvailableExpressionsInfo(function);

The above line generates files that perform the available expressions analysis and these files are included as header files in the main optimization program. availableexprs can then be passed as a parameter to functions that require available expressions information to be computed.

Liveness
liveness: findLivenessInfo(function);
The above line generates files that perform the liveness analysis and these files are included as header files in the main optimization program. liveness can then be passed as a parameter to functions that require available expressions information to be computed.

For each of the analyses, a data structure is created which is a map from the function name to a map of instructions in every function along with the respective dataflow analysis results at every instruction. Fig. 2 shows the generated structures when availableexprs is used.

```c++
struct std::vector<Expression*> expatt; // set of expressions
std::map<Instr, expatt> AVATT;     // at every instruction
std::map<std::string, itoE> AVExF; // for every function
```

Fig. 2. Data Structures Generated to Store Available Expressions

4) Sub-blocks:
The iterate sub-block: The iterate sub-block requires the compiler developer to specify an iterable along with it. The code that is inserted within the iterate block will be executed on every instance of the iterable in the program. The iterables allowed with the iterate sub-block are function, basicblock, loop and instruction. Listing 6 shows the use of the iterate sub-block where the iterable is a basic block. The code within this block contains another iterate block with the iterable instruction. This shows that for each basic block that is iterated through, every instruction within it is also iterated through. Listing 7 shows the respective translation in LLVM C++.

Listing 6. The iterate Sub-block
```c++
iterate : basicblock
{ iterate : instruction
}
```

Listing 7. LLVM C++ Translation of The iterate Sub-block
```c++
for (Function::iterator FI=F.begin(),
FE=F.end(); FI!=FE; ++FI)
{ BasicBlock* block = &*FI;
for (BasicBlock::iterator i=block->begin(),
e=block->end(); i!=e; ++i)
{ Instruction * I = &*i;
```

The repeat sub-block: The repeat sub-block allows the compiler developer to specify the number of times he or she requires a block of code to be generated. The syntax for this is seen in Listing 8 where n is the number of times the compiler developer wishes to perform the code within the IR.

Listing 8. The repeat Sub-block
```c++
repeat : n
{ }
```

5) Other Features: Other keywords have been made available specific to the chosen ten optimizations. The language supports C style comments. When multiple optimizations are specified in a program of the DSL, the order of specification of optimizations in the Co-Ex program is the order in which they are carried out on the IR. In other words, when the compiler developer wishes to carry out a sequence of different compiler optimizations on the IR, he or she must provide the specifications for them in the same order that he or she requires them to be performed. Following are some features of the DSL along with their translations:
- **precondition**: Certain functions may need preconditions before they are performed. When this keyword is passed as a parameter to a function, the code of the translation of the function is placed within an if block in LLVM C++ and the statement following precondition dictates the condition in the if block.
- **dag**: This keyword represents a directed acyclic graph of expressions that is used for local optimizations.
- **reversepostorder**: This keyword indicates that the traversal needs to be performed in reverse post order. Its translation is shown in Listing 9.

Listing 9. Translation for reversepostorder
```c++
DenseMap<Const DomTreeNodes*, unsigned> RPOOrdering;
```
ReversePostOrderTraversal<Function *> RPOT(&F);
unsigned Counter = 0;
for (auto &B : RPOT)
{
    auto *Node = DT->getNode(B);
    assert(Node && "RPOandDominatorTreeShouldHaveSameReachability");
    RPOOrdering[Node] = ++Counter;
}
for (auto &B : RPOT)
{
    auto *Node = DT->getNode(B);
    if (Node->getChildren().size() > 1)
    {
        std::sort(Node->begin(), Node->end(),
                   [&RPOOrdering](const DomTreeNode *A, const DomTreeNode *B)
                   {
                       return RPOOrdering[A] < RPOOrdering[B];
                   });
    }
}
auto DFI = df_begin(DT->getRootNode());
for (auto DFE = df_end(DT->getRootNode()); DFI != DFE; ++DFI)
{
    BasicBlock *B = DFI->getBlock();
    const auto &BlockRange = assignDFSNumbers(B, ICount);
    BlockInstRange.insert({B, BlockRange.first});
    ICount += BlockRange.second - BlockRange.first;
}

• makeDAG() : This function creates a DAG to hold binary expressions. Its translation is as shown in Listing 10.

Listing 10. Translation for makeDAG()
struct DAGInfoNode
{
    int binoperator;
    InstrList instructions;
    DAGInfoNode *next;
};
class DAGInfoList
{
    DAGInfoNode *first;
    public:
    DAGInfoList()
    {
        first = NULL;
    }
};
Instruction* findInDAG(int opCode, Instruction *newi)
{
    DAGInfoNode *searchcurr = curr->
    instructions.getHead();
    while (searchcurr!=NULL)
    {
        if (searchcurr->I->getOperand(0)==newi->
            getOperand(0)&
            &searchcurr->I->
            getOperand(1)==newi->
            getOperand(1))
            return searchcurr->I;
        searchcurr = searchcurr->next;
    }
    addNewInstr(curr,newi);
    return NULL;
}
auto DFI = df_begin(DT->getRootNode());
for (auto &B : RPOT)
{
    auto *Node = DT->getNode(B);
    assert(Node && "RPOandDominatorTreeShouldHaveSameReachability");
    RPOOrdering[Node] = ++Counter;
}
for (auto &B : RPOT)
{
    auto *Node = DT->getNode(B);
    if (Node->getChildren().size() > 1)
    {
        std::sort(Node->begin(), Node->end(),
                   [&RPOOrdering](const DomTreeNode *A, const DomTreeNode *B)
                   {
                       return RPOOrdering[A] < RPOOrdering[B];
                   });
    }
}
auto DFI = df_begin(DT->getRootNode());
for (auto DFE = df_end(DT->getRootNode()); DFI != DFE; ++DFI)
{
    BasicBlock *B = DFI->getBlock();
    const auto &BlockRange = assignDFSNumbers(B, ICount);
    BlockInstRange.insert({B, BlockRange.first});
    ICount += BlockRange.second - BlockRange.first;
}

• Optimization Specific Keywords : Following are the keywords, thresholds and routines provided for different optimizations in the prototype. The thresholds that are provided make the optimizations configurable by the compiler developers. The detailed explanation of these keywords can be found in the Co-Ex user manual present at https://github.com/LLVM-Co-Ex/blob/master/UserManual.pdf

1) Function Inlining :
no Lines, no Uses, makeInline(function, thresholds), removeInlinedFunctions()
2) Constant Folding and Propagation : foldpropagate(instruction, precondition); foldable(instruction)
3) Dead Code Elimination : deadcodeelim()
4) Local Common Subexpression Elimination : noExprs, lcs(basicblock, dag, thresholds)
5) Induction Variable Simplification : indvarsimplify(loop)
6) Global Value Numbering : valuenumber(instruction), valuateInstructions(function)
7) Loop Invariant Code Motion : loopinvariants, motionstmts, findLoopInvariantStatements(loop, reachingdefs), findMotionCandidates(loop, loopinvariants), applyCodeMotion(loop, motionstmts)
8) Tail Recursion Elimination : tailcallelim(function, precondition), hasRecursiveTailCall(function)
9) Loop Unrolling : unrollLoop(loop, noLines)
10) Global Common Subexpression Elimination : noExpr, gcse(function, dag, thresholds)

Listing 12. A Sample Co-Ex Program for Loop Invariant Code Motion

```
op : LoopInvariantCodeMotion
  runOn(module);
  iterate : function
    reachingdefs : findReachingDefinitionsInfo(function);
    iterate : loop
      loopinvariants : findLoopInvariantStatements(loop, reachingdefs);
      motionstmts : findMotionCandidates(loop, loopinvariants);
      applyCodeMotion(loop, motionstmts);
  }
```

Listing 13. Sample from Code Escalation Generated On Compiling Co-Ex Program in Listing 12

```
virtual bool runOnFunction(Function &F)
{
  bool modified = false;
  LoopInfo &LI =
    getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
  assignIndicesToInstrs(&F);
  for (LoopInfo::reverse_iterator I = LI.begin(), E = LI.end(); I != E; ++I)
    addLoopIntoQueue("1");
    while (!LQ.empty())
      {
        Loop* L = LQ.back();
        if (L->getLoopPreheader() == NULL)
          return false;
        set<Value*> loopInvariantStatements =
          computeLoopInvariantStatements(L,F);
        SmallVector<BasicBlock*, 0> ExitBlocks;
        L->getExitBlocks(ExitBlocks);
        DominatorTreeWrapperPass *DTWP =
          getAnalysisIfAvailable<DominatorTreeWrapperPass>();
        DominatorTree *DT = DTWP ? DTWP->getDomTree() :
          nullptr;
        set<Value*> motionCandidates;
        for (std::set<Value*>::iterator i1 =
          loopInvariantStatements.begin(); i1 !=
          loopInvariantStatements.end(); ++i1)
          {
            Instruction *I = static_cast<Instruction*>("i1");
            bool dominatesCheck = true;
            for (int i2=0; i2<ExitBlocks.size(); i2++)
              if ((DT->dominates(I->getParent(), ExitBlocks[i2]))
                dominatesCheck = false;
        }
        }
```

c. LLVM Specific Compiler for Co-Ex

A compiler that is specific to LLVM has been developed for Co-Ex which translates programs written in Co-Ex to LLVM C++ code on compilation, i.e., once a program in Co-Ex is compiled using this compiler, appropriate LLVM C++ code is generated which serves as an LLVM pass (or multiple passes). The compiler therefore parses the Co-Ex code, checks for syntax and semantic errors and if there are no errors, it goes on to construct an abstract syntax tree, which when traversed produces LLVM C++ code. The compiler developer will thus have to compile the Co-Ex program to generate a code escalation that will work on LLVM IR. This prototype implementation has been carried out for ten optimizations at a high level. Listing 12 shows a sample Co-Ex program for performing loop invariant code motion and listing 13 shows the respective translation to LLVM C++ for this Co-Ex program.

4. RESULTS AND CONCLUDING REMARKS

It is seen from our work that Co-Ex has simplified the work of compiler designers by allowing a simple and efficient manner of specifying compiler optimizations. A set of ten reusable modules have been developed in LLVM C++ each of them performing a compiler optimization. These modules have been tested on select test cases (LLVM IR) that are available in the LLVM suite and it has been observed that they optimize the LLVM IR as desired. A prototype of Co-Ex has been implemented which is evidently easier to understand and program in as compared to LLVM compiler infrastructure. It is seen that optimizations that are written in thousands of lines of code using the LLVM infrastructure can be written in tens of
representations also. Table 1 depicts the performance of the compiler escalations that have been generated from Co-Ex programs.

5. FUTURE ENHANCEMENTS

The prototype of Co-Ex that we present is a basic one at a high level of abstraction. This implies that for every optimization, the functions designed in Co-Ex cover a large part of the algorithm, that is, a large chunk of the generated LLVM C++ code is represented by only a single function in Co-Ex. Further versions of Co-Ex could introduce functions that are more specific to the algorithms for different optimizations. This would mean that these routines represent smaller steps of the algorithm. This provides more flexibility to compiler developers by allowing them to combine different steps of different algorithms in the manner that they choose.

Also, the current version of Co-Ex targets only ten optimizations, each with varying levels of complexities. This list includes loop optimizations, local optimizations and global optimizations. This list could be expanded to more compiler optimizations. To implement a new optimization, existing routines for dataflow analyses, the different sub-blocks and other such keywords could be utilized. The developed Co-Ex compiler is restricted to generate LLVM C++ code that will work only on LLVM IR. Different compilers can be developed for Co-Ex which would generate code specific to other intermediate formats as well.

REFERENCES


<table>
<thead>
<tr>
<th>Optimization</th>
<th>IR (no. of lines)</th>
<th>Optimized IR (no. of lines)</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Invariant Code Motion</td>
<td>17</td>
<td>20</td>
<td>0.224</td>
</tr>
<tr>
<td>Global Value Numbering</td>
<td>32*</td>
<td>32*</td>
<td>0.004</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>20</td>
<td>15</td>
<td>0.001*</td>
</tr>
<tr>
<td>Local Common Subexpression Elimination + Dead code Elimination</td>
<td>95</td>
<td>46</td>
<td>0.002</td>
</tr>
<tr>
<td>Function Inlining + Global Value Numbering</td>
<td>80</td>
<td>100</td>
<td>0.005</td>
</tr>
</tbody>
</table>

*These values are irrelevant to the optimizations performed in these cases.