Approximate Dependence Analysis

Final Project Report

CS6383: Introduction to Compiler Engineering

December 22, 2020

Authors

Anilava Kundu (CS20MTECH01002)

Kuldeep Gautam (CS20MTECH01004)

Mentors

Utpal Bora

Dr. Ramakrishna Upadrasta

Acknowledgment

We would like to thank our faculty advisor, **Dr. Ramakrishna Upadrasta** of the CSE Department at IIT Hyderabad, for his constant guidance and support throughout the project.

We would also like to thank our mentor **Utpal Bora** for constantly guiding us throughout the project and helping us out at various stages of it.

Authors

Anilava Kundu (CS20MTECH01002)

Kuldeep Gautam (CS20MTECH01004)

Table of Contents

1. Project Overview 3
2. Basic Terminology 3
3. Project Goals 5
4. Project Progress 5
5. Implementation Details 7
6. Deliverables 11
7. Project Status 15
8. Conclusion 15
9. References 16

Project Overview

As we know OpenMP is a parallel programming construct which aims at providing the functionality of parallelism in the code with the help of a number of pragmas, called OpenMP constructs, which marks a section of the code as to be taken into consideration for parallelising it. Since, we are talking about parallel programming here, we know that it is definitely possible that a shared variable is accessed by a number of threads or processes, resulting in a data race. To deal with this, data dependence analysis seems to be a good direction to verify the data dependencies and then using it to verify the possibility of data races in that section of the code containing the dependencies. The idea of the project is to use LLVM approximate dependence analysis to find such data dependencies to verify the possible data races in the program.

Basic Terminology

The following pointers provides an overview of the basic terminologies to get a better understanding of the underlying concepts:

1. **Data Dependence**
   A data dependence in a program or a set of instructions is a situation that occurs when two or more than two instructions in a program are trying to access the same memory location for reading and writing operations as modification in the value stored at a memory location if accessed in another instruction, may result in incorrect results. In parallel programming, when a number of threads or processes access a same memory location and try to modify it then at the end of execution it becomes vague that which particular process/thread actually updated the location, resulting in a ***Data Race.***

2. **Dependence Analysis**

   In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

3. **Types of Dependences**

   Depending on the type of conflicts between read and write operations, types of data dependences are as follows:

   a. ***RAW or Read After Write or Flow Dependence***[8]

      A flow dependency occurs when an instruction depends on the result of a previous instruction

   b. ***WAR or Write After Read or Anti-Dependence***[8]

      An anti-dependency occurs when an instruction requires a value that is later updated.

   c. ***WAW or Write After Write or Output Dependence***[8]

      An output dependency occurs when the ordering of instructions will affect the final output value of a variable.

4. **Representations of Dependence**

   A data dependence can be represented in a number of ways, namely:

   a. Dependence Vectors
   b. Dependence Levels
   c. Dependence Cone
   d. Dependence Direction Vectors (DDV)
   e. Dependence Polyhedra.

Project Goals

The goal of this project is to detect the possible data races in OpenMP parallel constructs using the already existing LLVM's Approximate Dependence Analysis Pass[5] (as per the initial plan). To detect the data races, the idea here is to determine the data dependence among the instructions inside a loop nest. Hence goals can be defined as follows:

1. Detecting data dependences inside a loop nest.
   This can be done using the approximate dependence analysis pass, already implemented in LLVM. While going through some of the examples and checking the detected data dependences in various examples, the already existing dependence analysis pass does not prove to be beneficial as it was not correctly determining the data dependences.

2. Extending the detection of data dependences to detect the data races in OMP parallel constructs.

Project Progress

This section describes the study and overall progress made during the project:

1. Studied about:
   a. LLVM loop terminology and canonicalization [2]
      Canonicalization of a loop intends to convert a loop into a canonical form makes it easier to perform optimizations on it and also to create a standard representation of a loop. Loops can be canonicalized and once a standard structure has been created after canonicalization, it can help in getting hold of array subscripts easily using SCEV, which

has been used in our implementation part.

There can be many ways to canonicalize a loop that LLVM has to offer in terms of optimizations, namely:

- ➢ Loop Simplify (opt pass: *-loop-simplify*),
- ➢ Loop Rotate (opt pass: *-loop-rotate*),
- ➢ Loop Closed SSA (opt pass: *-lcssa*)

b. Representation of dependences and respective examples [3]

c. Basic OpenMP constructs (particularly omp parallel for) and examples [4,5]

2. Wrote a basic LLVM pass (based on already existing approximate dependence analysis pass in LLVM) to check data dependency in a few examples. Following observations were made:
    a. The default LLVM approximate dependence analysis pass outputs a different type of dependency as compared to what really should be shown.

```
for (int j=0; j<m; j++)
{
for (int i=0; i<n; i++)
{
      x = A[i-1][j];
     A[i][j+2] = x+1;
}
}
```

```
Printing analysis 'Dependence Analysis' for function 'main':
Src:  %0 = load i32, i32* %arrayidx5, align 4, !dbg !32 --> Dst:  %0 = load i32, i32* %arrayidx5, align 4, !dbg !32
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx5, align 4, !dbg !32 --> Dst:  store i32 %add, i32* %arrayidx10, align 4, !dbg !37
  da analyze - anti [<> >]!
```

This is reported as an anti dependence whereas it should be reported as loop-carried flow dependence with a consistent distance vector of [2 1 \| 0].

b. The default LLVM dependence analysis pass fails to output the distance vector for a few cases and can only output the direction in such cases for the set of same examples from the website above. The website shows both distance and direction vectors for the example. But when ran with existing LLVM DA pass, such distance vectors were not reported.

Implementation Details

LLVM Version: 10.0.1
Operating System: Ubuntu 18.04

The implementation is largely focused on loop-carried dependencies viz array accesses within loops. To remove the dependency of the off-shelf dependency pass that is already there in LLVM we tried to implement a basic version of the same. The implementation largely uses the Scalar Evolution representation of LLVM to reduce array access expressions within loops.

Scalar Evolution

Scalar Evolution refers to the change of scalar variables over iteration of loops. Scalar Evolution uses symbolic technique to mathematically represent the expression for change of scalars.

Consider the following loop example:

```
for(int i=0 ; i< m ; i++)
     { t=t+k ; }
```

For every iteration of the loop the scalar value t is being incremented by k which is the evolution of the scalar value t. If we look carefully the value of t changes as **k,2k,3k….mk** with the loop terminating. If t were to be used after the loop the last value t would hold, that is **mk**. Scalar Evolution analysis is hence used to perform

such reductions. The LLVM SCEV is a practical implementation of Scalar evolution.

Steps:

1. ***Getting all Loops***

   The implementation starts off with extracting all the loops using the **LoopInfoWrapperPass** using getAnalysisUsage. Once all the loops are gathered we can access all the basic blocks within the loops

2. ***Iterating Basic Blocks for Instructions***

   We loop through all the instructions within the basic blocks present within the loops and pick instructions that may read or write to memory (store or load instructions). For every instruction **i** that reads or writes memory locations we run another loop to get all instructions **j** which is not i and may read or write to memory

3. ***Getting Array Accesses***

   Once we get all the pairs of instructions that read or write to memory we check if they are GEP instructions or not if so we convert them to SCEV forms using **ScalarEvolutionWrapperPass** with getAnalysisUsage to get a reduce expression for array accesses

4. ***Checking for Similar Array Accesses***

   With array accesses in SCEV form we cast it to SCEVAddRecExpr and parse it to get the array that is being accessed. Once this is obtained we do a simple check to see if the array accesses are on the same array or not.

5. ***Getting Iteration Distance and loop Level***

   If the array accesses are similar we cast the index operand of the GEP instruction into SCEVAddRecExpr to extract the subscripts of the array and also get the level of the loop the array access is associated with. All of this info is stored in a separate structure that is implemented called **AccessInfo** that is later used to find dependencies.

6. ***Getting Dependency Type***

   If the loop levels of the array accesses are the same we check the iteration distance of the array accesses obtained from the last step. We check if the iteration of a read instruction(load) is greater than a write operation (store) every array element will be written to first and then will be read later which is a **RAW** dependency.

   Similarly for the other way around we get a WAR dependency.

Limitations :

1. The above implementation works only for SIV (Single Induction Variable) and only for array access of the same loop. This implementation will also not work for MIV (multiple Induction Variable) or Coupled loop accesses.

2. The implementation only detects dependencies of type RAW or WAR. WAW is not supported as of now.

3. Aliases of arrays are also not handled which might be accessed inside loops.

4. Although the list of limitations are long the implementation proposed here can be extended to handle these which can be one of the future work for this project.

Pseudocode

```
-------------------------------------------------------------------------------
L <- getLoops()
for all BasicBlocks B  in L:
  for all instructions i  within B:
    if ( i.WriteorReadMemory and isGEPInst(i)):
      for all instructions j and i! =j  within B:
        if ( j.WriteorReadMemory and isGEPInst(j)):
          arrAccess1 <- getSCEV(getPointerOperand(i))
          arrAccess2 <- getSCEV(getPointerOperand(j))

          //Checking if array accesses are similar or not
          if(arrAccess1 == arrAccess2):
            arrAccess1.level= getLoopLevel()
            arrAccess2.level= getLoopLevel()
            arrAccess1.IterationDistance = getDistance()
            arrAccess2.IterationDistance = getDistance()

            //Same loop level
            if(arrAccess1.level == arrAccess2.level):

            //getType of instruction (Store or Load)
              arrAccess1.InstType = getType(i)
              arrAccess2.InstType = getType(j)

              if(arrAccess1.InstType == Store and arrAccess2.InstType == Load):
                if(arrAccess1.IterationDistance>arrAccess2.IterationDistance):
                  dependency="WAR"

                if(arrAccess1.IterationDistance<arrAccess2. IterationDistance):
                  dependency="RAW"

              else if(arrAccess2.InstType == Store and arrAccess1.InstType==Load):
                if(arrAccess1.IterationDistance>arrAccess2.IterationDistance):
                  dependency="RAW"

                if(arrAccess1.IterationDistance<arrAccess2. IterationDistance):
                  dependency="WAR"

-------------------------------------------------------------------------------
```

Deliverables

We provide a basic implementation of the proposed methodology, to determine the data dependence, as an LLVM pass. We show the differences between the already existing LLVM dependence analysis pass and what we have implemented on our own and show the correctness of our pass over the already existing one using some test programs. **Below are the differences:**

a. **test1.c**

```c
#include <stdio.h>
int main()
{
   int c = 2;
   int arr[100];
   for(int i=0; i<100; i++)
   {
       arr[i] = arr[i + 1] + c;
   }

   for (int i = 0; i < 100; i++)
   {
       c = c*2;
       arr[i] = c;
       arr[i-2] = arr[i];
   }

   return 0;
}
```

Expected Data Dependence: RAW at line 8 and 15

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/Approximate Dependency$ ../llvm-project/build/bin/opt -da -analyze opt_test1.ll
Printing analysis 'Dependence Analysis' for function 'main':
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28
  da analyze - consistent anti [1]!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42
  da analyze - anti [|<]!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43
  da analyze - input [|<]!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46
  da analyze - anti [|<]!
Src:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28 --> Dst:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28
  da analyze - none!
Src:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28 --> Dst:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42
  da analyze - output [|<]!
Src:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28 --> Dst:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43
  da analyze - flow [|<]!
Src:  store i32 %add1, i32* %arrayidx3, align 4, !dbg !28 --> Dst:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46
  da analyze - output [|<]!
Src:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42 --> Dst:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42
  da analyze - none!
Src:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42 --> Dst:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43
  da analyze - consistent flow [0|<]!
Src:  store i32 %mul, i32* %arrayidx9, align 4, !dbg !42 --> Dst:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46
  da analyze - consistent output [2]!
Src:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43 --> Dst:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43
  da analyze - none!
Src:  %1 = load i32, i32* %arrayidx11, align 4, !dbg !43 --> Dst:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46
  da analyze - consistent anti [2]!
Src:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46 --> Dst:  store i32 %1, i32* %arrayidx13, align 4, !dbg !46
  da analyze - none!
```

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/llvm-project/build$ ./bin/opt -load ./lib/Dependence.so -Dependence ../../Approximate\ Dependency/opt_test1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

RAW at line: 15 col: 20 --> line: 15 col: 18
RAW at line: 8 col: 18 --> line: 8 col: 16
```

b. test2.c

```c
#include <stdio.h>
int main()
{
    int c = 100;
    int arr[100];
    int brr[100];
    for(int i = 0; i < 100; i++)
    {
        arr[i] = arr[i-1] + c;
        for(int j = 0; j < 100; j++)
```

```
        {
            brr[j] = arr[i];
            brr[j-2] = brr[j] + c;
        }
    }
    return 0;
}
```

Expected Data Dependence: WAR at line 9 and RAW at line 13

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/Approximate Dependency$ ../llvm-project/build/bin/opt -da -analyze opt_test2.ll
Printing analysis 'Dependence Analysis' for function 'main':
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30
  da analyze - consistent anti [-1]!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35
  da analyze - consistent input [-1]!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !27 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - none!
Src:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30 --> Dst:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30
  da analyze - none!
Src:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30 --> Dst:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35
  da analyze - consistent flow [0|<]!
Src:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30 --> Dst:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39
  da analyze - none!
Src:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30 --> Dst:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40
  da analyze - none!
Src:  store i32 %add, i32* %arrayidx2, align 4, !dbg !30 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - none!
Src:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35 --> Dst:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35
  da analyze - consistent input [0 S]!
Src:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35 --> Dst:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39
  da analyze - none!
Src:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35 --> Dst:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40
  da analyze - none!
Src:  %1 = load i32, i32* %arrayidx7, align 4, !dbg !35 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - none!
Src:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39 --> Dst:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39
  da analyze - consistent output [S 0]!
Src:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39 --> Dst:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40
  da analyze - consistent flow [S 0|<]!
Src:  store i32 %1, i32* %arrayidx9, align 4, !dbg !39 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - consistent output [S 2]!
Src:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40 --> Dst:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40
  da analyze - consistent input [S 0]!
Src:  %2 = load i32, i32* %arrayidx11, align 4, !dbg !40 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - consistent anti [S 2]!
Src:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44 --> Dst:  store i32 %add12, i32* %arrayidx15, align 4, !dbg !44
  da analyze - consistent output [S 0]!
```

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/llvm-project/build$ ./bin/opt -load ./lib/Dependence.so -Dependence ../../Approximate\ Dependency/opt_test2.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

WAR at line: 9 col: 18 --> line: 9 col: 16
RAW at line: 13 col: 24 --> line: 13 col: 22
```

c. test4.c

```c
#include <stdio.h>
int main()
{
    int a;
    int c = 2;
    int arr[100];
    for (int i = 0; i < 100; i++)
    {
        a = arr[i+1] + c;
        arr[i-2] = c*10;
    }
    return 0;
}
```

Expected Data Dependence: RAW at line 9

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/Approximate Dependency$ ../llvm-project/build/bin/opt -da -analyze opt_test4.ll
Printing analysis 'Dependence Analysis' for function 'main':
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25
  da analyze - none!
Src:  %0 = load i32, i32* %arrayidx, align 4, !dbg !25 --> Dst:  store i32 %mul, i32* %arrayidx3, align 4, !dbg !31
  da analyze - consistent anti [3]!
Src:  store i32 %mul, i32* %arrayidx3, align 4, !dbg !31 --> Dst:  store i32 %mul, i32* %arrayidx3, align 4, !dbg !31
  da analyze - none!
```

```
lavo07@lavo07-Lenovo-ideapad-330-15IKB:/media/lavo07/lavo07/llvm-project/build$ ./bin/opt -load ./lib/Dependence.so -Dependence ../../Approximate\ Dependency/opt_test4.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

RAW at line: 9 col: 13 --> line: 10 col: 18
```

Project Status

Our project implementation is in the initial phase of development and could be further improved upon and had been extended to tackle more complex loop nests and data dependences but could not be done due to time constraints. We think that our **implementation covers about 20% to 25% of the overall work** that this project demands.

Conclusion

In this project, we started with an initial study on various aspects of what is required in the project which helped us understand what the project demands and how it needs to be done. Initially, we thought of using the approximate dependence analysis pass provided by LLVM itself, but that came out to be not performing on a number of examples and resulting in incorrect data dependences as well as absence of distance vectors, which should be printed (pointed out above in the screenshot attached). After discussing the problems with the mentor, we concluded that we should go for a custom implementation for the dependence analysis as explained above. **A very basic implementation has been created as a deliverable for this project which can be found in the Github repo** *Approximate-Dependence-Analysis*.

References

1. https://engineering.purdue.edu/~milind/ece573/2011spring/lecture-14.pdf
2. https://llvm.org/docs/LoopTerminology.html
3. https://sites.google.com/site/parallelizationforllvm/representing-dependences
4. https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf
5. https://www.openmp.org//wp-content/uploads/openmp-examples-4.5.0.pdf
6. https://dl.acm.org/doi/abs/10.1145/113446.113448?casa_token=kbw86MKvcdkAAAAA:3pQiYUJy58tycVRY1xORDly06ObP1O_EX2rAPJKifGqU210s2vKT86B2lEkJnVELvkJqCgDT1-x6
7. https://llvm.org/doxygen/DependenceAnalysis_8cpp_source.html
8. https://en.wikipedia.org/wiki/Data_dependency