Contents

1. Cyclomatic Complexity Analysis using LLVM/Clang

Extend the LLVM/Clang compiler to compute the cyclomatic complexity of all functions in each source file. The results should be written to a new file with the same name as the source file but with a `.cy` extension.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-cyclomatic-complexity) to enable the cyclomatic complexity analysis.
- Implement the functionality to compute the cyclomatic complexity of each function in the source file.
- Ensure that the analysis results are written to a ‘<source_file>.cy’ file.
- Each line of the .cy file should contain the function name and its cyclomatic complexity.
- Test the functionality on popular open-source C/C++ projects (e.g., LLVM itself, Linux Kernel, OpenSSL, etc.).
- Validate the correctness and performance of the implemented feature.

Expected Output

```
clang -analyze-cyclomatic-complexity example.c

# This should generate a file named example.cy with content like:

# function1 4

# function2 7

# ...
```

2. Static Analysis for Identifying Data Parallelism Using LLVM/Clang

Develop a static analysis tool using LLVM/Clang to identify functions in C/C++ code that exhibit data parallelism, i.e. performing the same operation on independent data elements. The tool should emit a diagnostic message for each function demonstrating data parallelism.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-data-parallelism) to enable the data parallelism analysis.
- Define what constitutes data parallelism (e.g., independent iterations of loops).
- Determine patterns and indicators of data parallelism in LLVM IR.
- Identify loops and determine if iterations are independent (i.e., no data dependencies).
- Emit diagnostics for functions with data parallelism, explaining the detected data parallelism.

Example(s)

Data Parallelism

```
void compute(float* data, int N) {
    for (int i = 0; i < N; ++i) {
        data[i] = sin(data[i]) +
cos(data[i]);
    }
}
```

No Data Parallelism

```
void process(float* data, int N) {
    for (int i = 0; i < N; ++i) {
        if (data[i] > 0) {
            data[i] = sqrt(data[i]);
        }
    }
}
```

Expected Output

Function 'compute' has data parallelism: - Independent loop iterations detected at line 42

3. Static Analysis for Assessing Computational Intensity Using LLVM/Clang

Develop a static analysis tool using LLVM/Clang to evaluate the computational intensity of functions, i.e. the ratio of arithmetic operations to memory operations. The tool should emit a diagnostic message for each function with high computational intensity.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-computational-intensity) to enable the computational intensity analysis.
- Define computational intensity and relevant indicators (e.g., arithmetic operations vs. memory operations).
- Determine patterns and indicators of computational intensity in LLVM IR.
- Count arithmetic and memory operations to evaluate intensity.
- Emit diagnostics for functions with high computational intensity, explaining the detected intensity.

Example(s)

High Computational Intensity

```
void compute(float* data, int N) {
    for (int i = 0; i < N; ++i) {
        data[i] = sin(data[i]) +
cos(data[i]);
    }
}
```

Low Computational Intensity

```
void initialize(float* data, int N) {
    for (int i = 0; i < N; ++i) {
        data[i] = 0.0f;
    }
}
```

Expected Output

```
Function 'compute' has high computational intensity:
 - High ratio of arithmetic operations to memory operations detected
```

4. Static Analysis for Detecting Regular Memory Access Patterns Using LLVM/Clang

Develop a static analysis tool using LLVM/Clang to identify functions in C/C++ code that exhibit regular memory access patterns, such as sequential access. The tool should emit a diagnostic message for each function with regular memory access patterns.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-regular-memory-access) to enable detection of regular memory access patterns.
- Define what constitutes regular memory access patterns (e.g., sequential access).
- Determine patterns and indicators of regular memory access in LLVM IR.
- Identify and evaluate memory access patterns for regularity.
- Emit diagnostics for functions with regular memory access patterns, explaining the detected access patterns.

Example(s)

Regular Memory Access Pattern

```
void compute(float* data, int N) {

    for (int i = 0; i < N; ++i) {

        data[i] = sin(data[i]) +
cos(data[i]);

    }

}
```

Irregular Memory Access Pattern

```
void scatter(float* data, float*
output, int* indices, int N) {

    for (int i = 0; i < N; ++i) {

        output[indices[i]] = data[i];

    }

}
```

Expected Output

Function 'compute' has regular memory access patterns:

- Sequential access detected at line 42

5. Function Cycle Count Analysis and Reporting Using LLVM/Clang

The goal of this project is to extend the LLVM/Clang compiler to compute the execution cycle count of all functions in each source file. Students will introduce a new compiler option to enable this functionality and ensure that the cycle count for each function is printed to the terminal upon compilation.

Key Tasks

- Introduce a new command-line option (e.g., -function-cycle-count) to enable the function cycle count analysis.
- Use LLVM's analysis passes to estimate the cycle count for each function, considering basic blocks, control flow, and any relevant hardware-specific details.
- Extend the Clang diagnostics to print the cycle count for each function to the terminal.

Expected Output

```
clang -function-cycle-count example.c

Function main: 120 cycles

Function foo: 45 cycles

Function bar: 78 cycles
```

6. Implement Clang identifiers to retrieve the mangled name and the fully qualified name of functions

Enhance the Clang compiler by introducing two new identifiers: __fq_func__ for retrieving the fully qualified name of the current function and __mangled_func__ for retrieving the mangled name of the current function, similar to the __func__ identifier.

Key Tasks

- Modify the Clang frontend to support the __fq_func__ identifier, ensuring that it retrieves the fully qualified name of the function, including namespaces and class names.
- Modify the Clang frontend to support the __mangled_func__ identifier, ensuring that it retrieves the mangled name of the function as generated by the LLVM backend.
- Write test cases to validate that __fq_func__ and __mangled_func__ work correctly in various contexts, including different namespaces and class hierarchies.

Example(s)

```
namespace Foo {
   class Bar {
      void myFunction() {
         std::cout << "Current function: " << __fq_func__ << std::endl;
         std::cout << "Mangled function: " << __mangled_func__ << std::endl;
      }
   };
}
```

Expected Output

Current function: Foo::Bar::myFunction

Mangled function: _ZN3Foo3Bar11myFunctionEv

7. Instruction Frequency Analysis Pass for LLVM/Clang

Develop a new analysis pass in the LLVM compiler framework that iterates over all functions in each program and classifies each instruction into predefined categories. The pass will then create a frequency table for each function, detailing the number of instructions in each category. This frequency table will be emitted into a file with the same name as the source file but with a '.ic' extension.

Instruction Categories

The instructions will be classified into the following categories:

- **Arithmetic**: Includes instructions for addition, subtraction, multiplication, division, remainder, etc.
- **Logical**: Includes bitwise operations like AND, OR, XOR, and shifting instructions.
- **Comparison**: Includes integer and floating-point comparisons (e.g., icmp, fcmp).
- **Memory**: Includes memory operations like load, store, allocation (alloca), and element access (getelementptr).
- **Control Flow**: Includes branching (br, condbr), multiway branching (switch), and phi-nodes (phi).
- **Function Call**: Includes function call instructions (call, invoke) and return instructions (ret).

Key Tasks

- Modify Clang to accept a new command-line option (e.g., -emit-instr-freq) that enables the analysis pass.
- Implement a new analysis pass in LLVM that iterates over all functions in the input program and classifies each instruction into one of the predefined categories.
- Maintain a frequency table for each function in the source file being compiled.
- For each source file compiled, emit the frequency table for each function into a '<source_file>.ic' file.
- Validate the functionality against a set of popular open-source C/C++ projects (e.g., LLVM itself, SQLite, Git, Cmake).

Example(s)

```
#include <stdio.h>

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

int main() {
    int x = 10;
    int y = 20;
    int z = add(x, y);
    if (z > 20) {
        printf("Result is greater than 20\n");
    } else {
        printf("Result is 20 or less\n");
    }
    return 0;
}
```

Expected Output

```
Function,Arithmetic,Logical,Comparison,Memory,Control Flow,Function Call
add,1,0,0,1,0,1
main,0,0,1,5,3,3
```

8. Instrumentation of C/C++ Code Using LLVM/Clang and PAPI for Performance Monitoring

Develop a tool that automatically instruments C/C++ code to gather Performance Monitoring Counters (PMC) data for each function using PAPI (Performance Application Programming Interface). The list of PMC events to be captured will be specified via command line. The tool should output the captured events per function per invocation in a CSV format, including the timestamps of the start and finish of each function's execution.

Key Tasks

- Install and configure PAPI.
- Develop a runtime library to initialize PAPI, start/stop event counters, and collect data. For example,

```
#include <stdio.h>
#include <stdlib.h>
#include <papi.h>

int main() {
    int event_set = PAPI_NULL;
    long long values[2];
    int retval;

    // Initialize PAPI library
    if ((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT)
{
        fprintf(stderr, "PAPI library init error!\n");
        exit(1);
    }

    // Create an event set
    if (PAPI_create_eventset(&event_set) != PAPI_OK) {
        fprintf(stderr, "Error creating event set!\n");
        exit(1);
    }

    // Add L1 data cache access event
    if (PAPI_add_event(event_set, PAPI_L1_DCA) != PAPI_OK) {
        fprintf(stderr, "Error adding event!\n");
        exit(1);
    }

    // Add L2 data cache access event
    if (PAPI_add_event(event_set, PAPI_L2_DCA) != PAPI_OK) {
        fprintf(stderr, "Error adding event!\n");
        exit(1);
    }

    // Start counting
    if (PAPI_start(event_set) != PAPI_OK) {
        fprintf(stderr, "Error starting PAPI!\n");
        exit(1);
    }

    // Code to measure
    for (int i = 0; i < 1000000; i++) {
        int x = i * i;
    }

    // Stop counting
    if (PAPI_stop(event_set, values) != PAPI_OK) {
        fprintf(stderr, "Error stopping PAPI!\n");
        exit(1);
    }

    // Output the counts
    printf("L1 data cache accesses: %lld\n", values[0]);
    printf("L2 data cache accesses: %lld\n", values[1]);

    // Cleanup
    PAPI_shutdown();

    return 0;
}
```

- Parse the input C/C++ code to identify function entry and exit points and insert calls to the runtime library. For example,

```
void foo() {
  runtime_function_entry("foo");
  // original function body
  runtime_function_exit("foo");
}
```

- Use LLVM's command-line parsing utilities to allow users to specify events (e.g. `-trace-papi-events="PAPI_TOT_INS,PAPI_L1_DCM"`)
- Store event counts and timestamps in a suitable data structure.
- On program exit, output the collected data to a CSV file.

Expected Output

```
function_name,start_timestamp,end_timestamp,PAPI_TOT_INS,PAPI_L1_DCM
foo,1623651123.123,1623651123.456,1000,20
bar,1623651123.789,1623651124.012,2000,30
```

9. Runtime Memory Access Analysis using LLVM/Clang

Develop a tool to instrument C/C++ code using LLVM/Clang, allowing it to track and report the heap memory addresses accessed by functions at runtime. The tool will print the function names along with the accessed memory locations (read/write) in CSV format. This will be enabled with a new Clang command-line option.

Key Tasks

- Introduce a new command-line option (e.g., -track-heap- access) to enable analysis of heap memory access.
- Implement a new LLVM function level pass
- Understand LLVM IR to identify load/store instructions and function boundaries.
- Instrument the LLVM IR to track function names.
- Instrument the LLVM IR track and log memory accesses (load/store instructions).
- Export logged data to a CSV file.

Expected Output

```
Function Name,Memory Address,Access Type,Source File,Line Number
main,0x7ffeea1b0af8,write,main.cpp,10
main,0x7ffeea1b0af8,read,main.cpp,11
processData,0x602000000010,write,main.cpp,22
processData,0x602000000018,read,main.cpp,23
calculate,0x602000000020,read,main.cpp,35
calculate,0x602000000028,write,main.cpp,36
```

10. LLVM Analysis Pass for Identifying Memory Access Patterns

Develop an LLVM analysis pass that identifies and reports load and store instructions within each function of a program. The pass will handle different memory access patterns, including direct loads/stores, pointer arithmetic, and nested structures. For each memory location being accessed, the pass will determine the type, size, frequency, and total amount of memory accessed. A comprehensive report will be generated for each source file, detailing the memory access patterns for each function.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-memory-access-patterns) to enable analysis of memory access patterns.
- Create data structures to store details of each memory access.
- Write a function level LLVM pass to detect `LoadInst` and `StoreInst`.
- Gather information about the memory being accessed.
- Implement handling of direct memory accesses.
- Implement handling for pointer arithmetic and nested structures.
- Use type analysis to determine the size of memory accesses.
- Track the frequency and total size of accesses for each memory location.
- Output the collected data into a readable report to standard output.

Expected Output

```
file1.c:function1()
myArray: 12 bytes (3 times)
myStruct.field1: 8 bytes (2 times)
myArray2: 4 bytes (1 time)
myStruct.field2: 8 bytes (2 times)

file2.c:function2()
myArray3: 24 bytes (3 times)
myStruct.field2: 16 bytes (4 times)
```

11. Function Scope Identifier Using LLVM/Clang Preprocessor

Develop a tool that extends the functionality of the Clang preprocessor to identify the function(s) that a given line range belongs to in C/C++ programs. This tool should preprocess the source files, supporting common preprocessor options, and handle include files and macro definitions.

Key Tasks

- Extend the Clang preprocessor to support a new command-line option with the ability to parse the argument specified as a comma-separated line range (e.g. -identify-scope-range=12-17,19-34).
  Hint: Ensure the tool supports standard preprocessor options along with the new option.
- Extend the Clang preprocessor to identify and track function boundaries using AST (Abstract Syntax Tree) or token analysis.
- Implement the logic to determine which function(s) a specified line range belongs to.
  Hint: Handle cases where a line range spans multiple functions.

Example(s)

```
# Basic usage
clang -E -identify-scope-range=12-17,19-34 test.c

# With include paths and macro definitions
clang -E -I./include -DDEBUG -identify-scope-range=12-17,19-34 test.c
```

Expected Output

```
Range 3-5:
  Function: foo
    Start Line: 5
    End Line: 10

Range 10-11:
  Function: bar
    Start Line: 12
    End Line: 16
```

12. Analyzing MPI Code for Data Gathering and Distribution Patterns using LLVM/Clang

Develop a tool using the LLVM/Clang compiler infrastructure to analyze C/C++ MPI (Message Passing Interface) code. The tool will identify instances where data is being gathered from multiple processes to a single process or distributed from a single process to multiple processes without using collective operations, providing insights into potential performance improvements.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-mpi-scatter-gather) to enable identification of MPI data gathering and distribution patterns.
- Identify typical patterns of data gathering and distribution in MPI programs.
- Implement analysis passes to identify non-collective data gathering and distribution patterns (i.e. no use of collectives such as `MPI_Gather`, `MPI_Scatter`, `MPI_Allgather`, or `MPI_Alltoall`).
- Implement detailed reporting functionality to provide actionable insights.

Example(s)

```
#include <mpi.h>
#include <stdio.h>

void gatherData(int* sendbuf, int* recvbuf, int count, int root, MPI_Comm comm) {
    MPI_Status status;
    int rank;
    MPI_Comm_rank(comm, &rank);

    if (rank == root) {
        for (int i = 0; i < count; ++i) {
            recvbuf[i] = 0;
        }
    }

    MPI_Sendrecv(sendbuf, count, MPI_INT, root, 0, recvbuf, count, MPI_INT, root,
0, comm, &status);
}

void distributeData(int* sendbuf, int* recvbuf, int count, int root, MPI_Comm
comm) {
    MPI_Status status;
    int rank;
    MPI_Comm_rank(comm, &rank);

    if (rank == root) {
        for (int i = 0; i < count; ++i) {
            recvbuf[i] = sendbuf[i];
        }
    }

    MPI_Sendrecv(sendbuf, count, MPI_INT, root, 0, recvbuf, count, MPI_INT, root,
0, comm, &status);
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data = rank;
    int gatheredData[4] = {0};

    gatherData(&data, gatheredData, 1, 0, MPI_COMM_WORLD);

    int sendbuf[4] = {1, 2, 3, 4};
    int recvbuf[4] = {0};

    distributeData(sendbuf, recvbuf, 4, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Expected Output

```
File: example.c

================================

Analysis of gatherData Function

================================

Pattern Detected: Non-collective Data Gathering

- Issue: Data is being gathered from all processes to the root process without using
collectives.

- Location: gatherData function, Line 5


Details:

- Code snippet:

  if (rank == root) {

      for (int i = 0; i < count; ++i) {

          recvbuf[i] = 0;

      }

  }


================================

Analysis of distributeData Function

================================

Pattern Detected: Non-collective Data Distribution

- Issue: Data is being distributed from the root process to all processes without using
collectives.

- Location: distributeData function, Line 14


Details:

- Code snippet:

  if (rank == root) {

      for (int i = 0; i < count; ++i) {

          recvbuf[i] = sendbuf[i];

      }

  }
```

13. Identification of MPI Reduction Opportunities in C/C++ Code Using LLVM/Clang

Identify and highlight potential MPI (Message Passing Interface) reduction operations in C/C++ code. These opportunities should be ones where multiple processes combine their data using operations like sum, min, max, etc. Provide a detailed report at completion.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-mpi-reduction) to enable identification of MPI reduction opportunities. The argument to the option specifies the reduction operations to identify
  For example,

  -analyze-mpi-reduction=sum

- Create an LLVM pass to implement pattern matching for common reduction operations in the LLVM IR.
- Use LLVM's analysis infrastructure to detect patterns like summing values across processes, finding the minimum or maximum, etc.
  Hint: Start with simpler patterns like summation
- Once a pattern is detected, generate a report detailing the opportunity.

Example(s)

```c
#include <mpi.h>
#include <stdio.h>

void manual_sum_reduction(int *sendbuf, int *recvbuf, int count, MPI_Comm comm) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    // Initialize recvbuf
    for (int i = 0; i < count; i++) {
        recvbuf[i] = 0;
    }

    // Manual reduction
    for (int i = 0; i < size; i++) {
        int temp[count];
        MPI_Recv(temp, count, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
        for (int j = 0; j < count; j++) {
            recvbuf[j] += temp[j];
        }
    }

    // Send data to all processes
    for (int i = 0; i < size; i++) {
        MPI_Send(recvbuf, count, MPI_INT, i, 0, comm);
    }
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int sendbuf[1] = {rank + 1}; // Example data to be reduced
    int recvbuf[1];

    manual_sum_reduction(sendbuf, recvbuf, 1, MPI_COMM_WORLD);

    printf("Process %d received reduced result %d\n", rank, recvbuf[0]);

    MPI_Finalize();
    return 0;
}
```

Expected Output

```
File: example.c
----------------------------------------
1. Function: manual_sum_reduction
   Reduction Type: Sum
   Identified Pattern: Manual summing of received values in recvbuf
   Location: example.c:15-21

   Code Snippet:
   15:     for (int i = 0; i < size; i++) {
   16:         int temp[count];
   17:         MPI_Recv(temp, count, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
   18:         for (int j = 0; j < count; j++) {
   19:             recvbuf[j] += temp[j];
   20:         }
   21:     }

   Details:
   - MPI_Recv followed by summing of received values into recvbuf.

2. Function: manual_sum_reduction
   Communication Pattern: Send data to all processes
   Location: example.c:23-25

   Code Snippet:
   23:     for (int i = 0; i < size; i++) {
   24:         MPI_Send(recvbuf, count, MPI_INT, i, 0, comm);
   25:     }

   Details:
   - Broadcasting the reduced result to all processes.
----------------------------------------
Total reduction opportunities identified: 2
```

14. Analyzing Uniform Participation of MPI Processes Using LLVM/Clang

Statically analyze MPI-based C/C++ code and identify patterns of uniform participation among MPI processes. Specifically, detect scenarios where multiple individual sends or receives involve the same set of processes.

Key Tasks

- Introduce a new command-line option (e.g., -analyze-mpi-uniform-participation) to enable analysis of uniform participation of MPI processes.
- Create an LLVM pass to parse and analyze the IR of MPI code, detecting MPI communication calls (i.e. `CallInst`, `MPI_Send` and `MPI_Recv`).
- Extract relevant information such as process ranks and message tags.
- Store the detected MPI call details in a data structure for further analysis.
- Implement analysis logic to identify uniform participation patterns (e.g. check if the same set of ranks are involved in multiple MPI_Send or MPI_Recv calls).
- Format the analysis results into a readable report and print to standard output on completion.

Example(s)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        int data = 100;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        int data;
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data: %d\n", data);
    }

    MPI_Finalize();
    return 0;
}
```

Expected Output

```
[INFO] Detected MPI Send: comm=MPI_COMM_WORLD, tag=0, rank=0

[INFO] Detected MPI Recv: comm=MPI_COMM_WORLD, tag=0, rank=1


[INFO] Analyzing Uniform Participation Patterns...

[INFO] Uniform Participation Detected in Comm MPI_COMM_WORLD with Tag 0 involving Ranks: 0, 1




Uniform Participation Report:

------------------------------------

- Communicator: MPI_COMM_WORLD

- Tag: 0

- Participating Ranks: {0, 1}


This indicates that both MPI_Send and MPI_Recv operations with tag 0 in communicator
MPI_COMM_WORLD involve ranks 0 and 1.
```

References

1. https://llvm.org/docs/
2. https://clang.llvm.org/docs/
3. http://icl.utk.edu/papi/docs/
4. https://www.mpich.org/
5. https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/
6. https://www.mpi-forum.org/docs/
7. https://mpitutorial.com/
8. https://github.com/mpitutorial/mpitutorial/tree/gh-pages/tutorials