

Scalable Tools Workshop

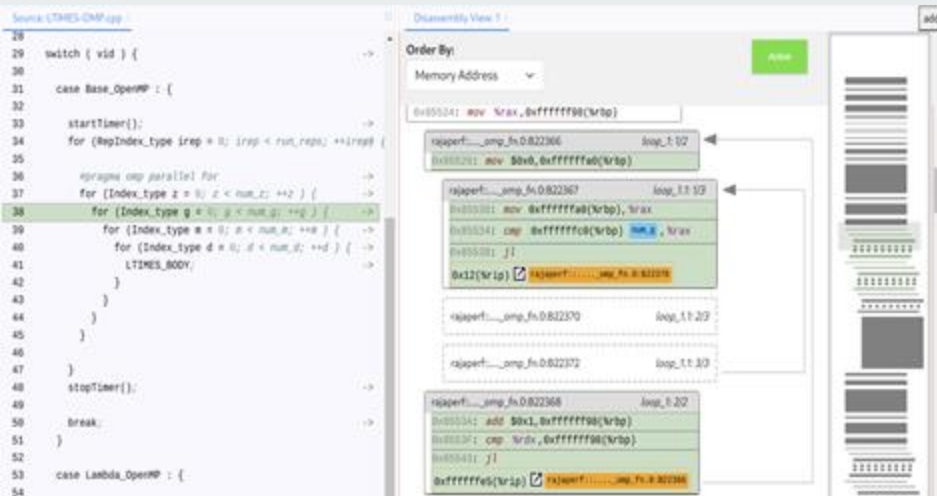


Interactive Visualization of Binary Code for Investigating Compiler Optimizations

Shadmaan Hye

Graduate Student, University of Utah

June 19, 2023



Problem: Labor Intensive Compilation Analysis

Multiple Compilers (gcc, intel, clang, ...)

Multiple Compilation Options (-O2, -O3, -funroll-loops, ...)

Multiple Optimizations (inlining, vectorization, loop unrolling, variable hoisting, ...)

Enable Code Developers to understand what optimization compiler is doing to the code through a visualization system



```
#include "DAXPY.hpp"
#include "RAJA/RAJA.hpp"

#include <iostream>

namespace rajaperf
{
namespace basic
{

void DAXPY::runSeqVariant(VariantID vid)
{
    const Index_type run_reps = getRunReps();
    const Index_type ibegin = 0;
    const Index_type iend = getRunSize();

    DAXPY_DATA_SETUP;

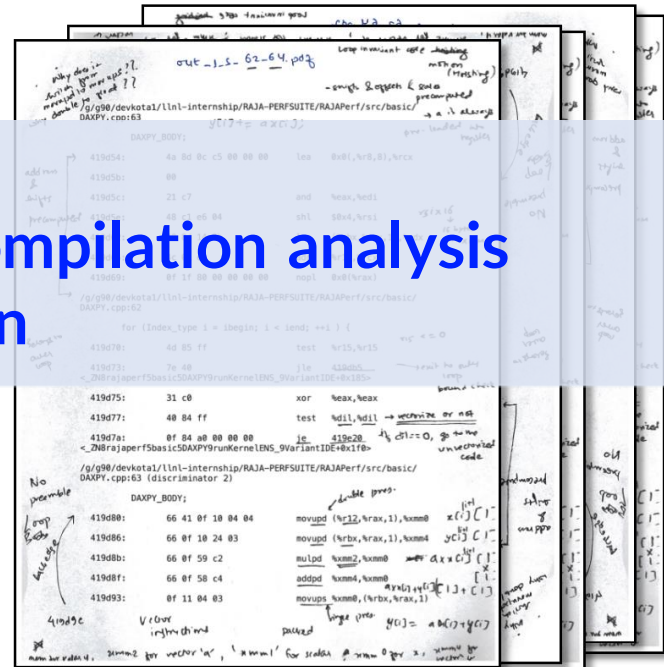
    auto daxpy_lam = [=](Index_type i) {
        DAXPY_BODY;
    };

    switch ( vid ) {

    case Base_Seq : {
```

Main Goal

To improve the human side of compilation analysis with visualization



Our Visualization Interface

Take binary file
compiled with debug
flag as input

File structures
shown here



Source code shown in
Source: [file.name] tab



Disassembly code
shown in order



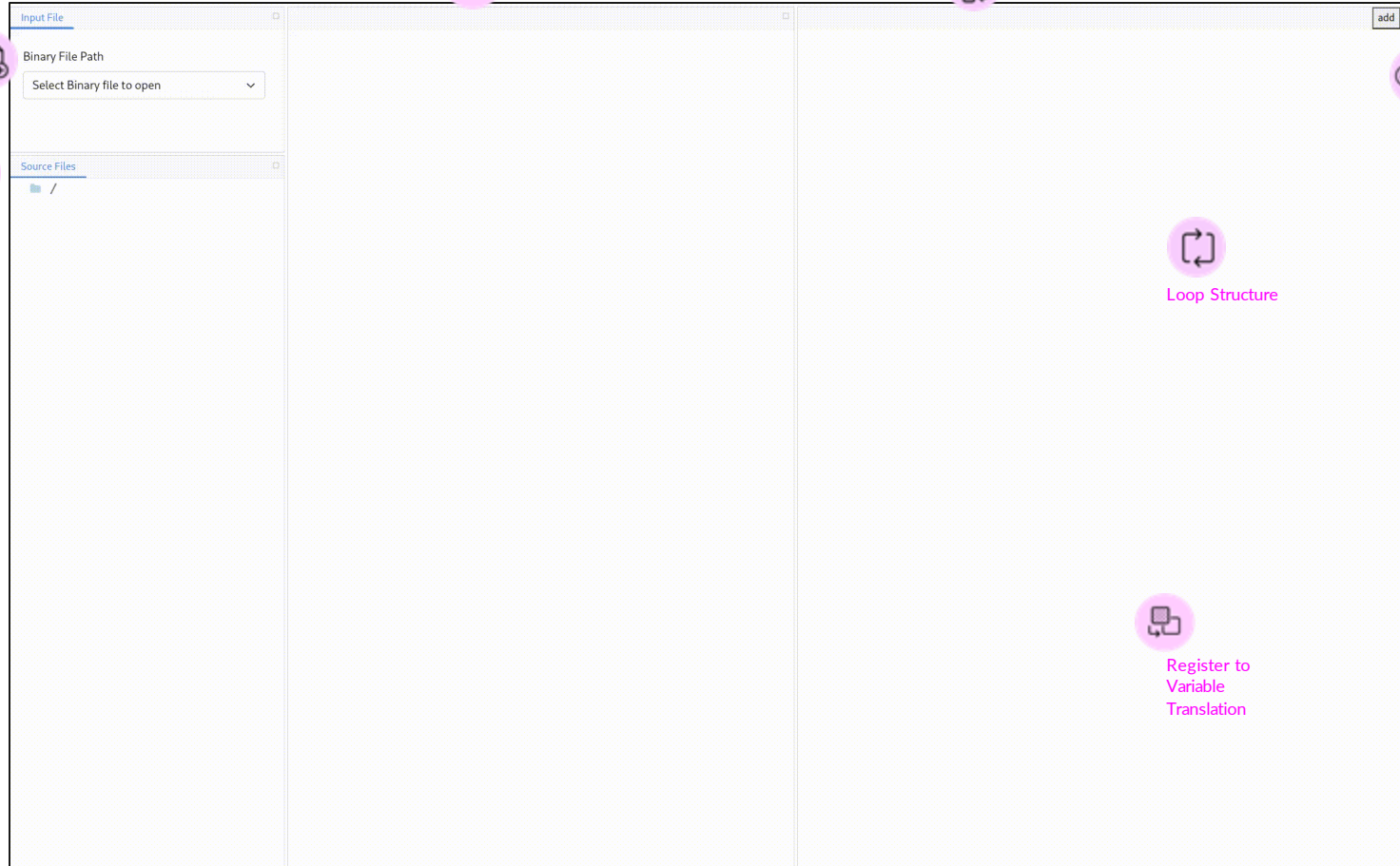
Disassembly
code and
current location
overview



Loop Structure



Register to
Variable
Translation



Disassembly View

No vertical space between blocks of one function

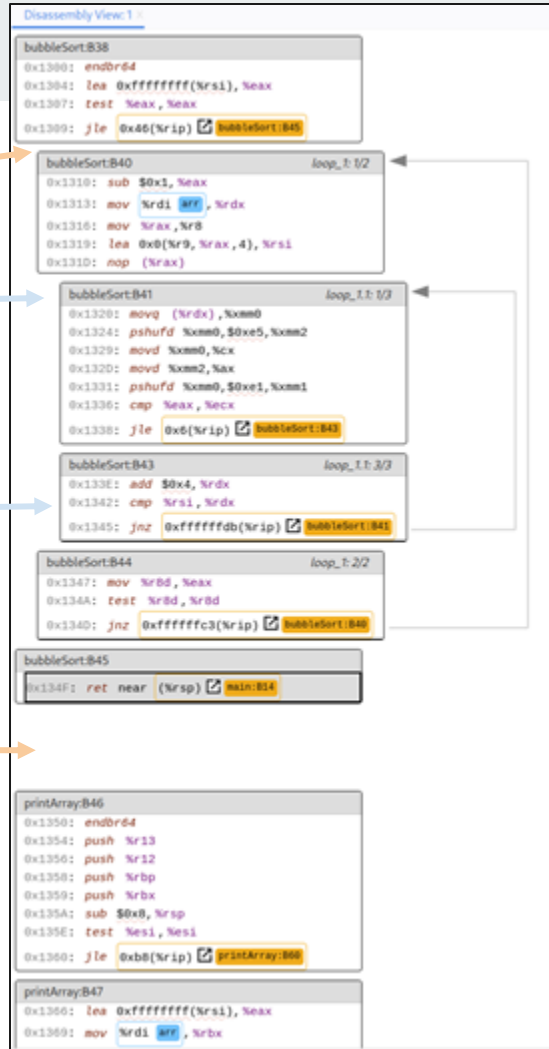
Block Header:

Function_name: B<Unique id>

Block Body:

Address: **operator** operand, operand

Vertical space between blocks of different function



Focus on Loops for Optimization

Programs spend the most time executing loops

Understanding which instructions are executing multiple times is vital

Focus on Loops

The image displays a side-by-side comparison of C++ source code and its x86-64 assembly disassembly for a bubble sort algorithm.

Source: bubble_sort.cpp

```
1 // C++ program for implementation
2 // of Bubble sort
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 // A function to implement bubble sort
7 void bubbleSort(int arr[], int n)
8 {
9     int i, j;
10    for (i = 0; i < n - 1; i++)
11    {
12        // Last i elements are already
13        // in place
14        for (j = 0; j < n - i - 1; j++)
15            if (arr[j] > arr[j + 1])
16                swap(arr[j], arr[j + 1]);
17    }
18    // Function to print an array
19    void printArray(int arr[], int size)
20    {
21        int i;
22        for (i = 0; i < size; i++)
23            cout << arr[i] << " ";
24        cout << endl;
25    }
26 }
27
28 // Driver code
29 int main()
30 {
31     int arr[] = { 5, 1, 4, 2, 8 };
32     int N = sizeof(arr) / sizeof(arr[0]);
33     bubbleSort(arr, N);
34     cout << "Sorted array: \n";
35     printArray(arr, N);
36     return 0;
37 }
```

Disassembly View: 1

Order By: Memory Address

bubbleSort:B39

```
0x1308: lea 0x4(%rdi),%r9
0x130F: nop
```

bubbleSort:B40 *loop_1: 1/2*

```
0x1310: sub $0x1,%eax
0x1313: mov %rdi,%rdx
0x1316: mov %rax,%r8
0x1319: lea 0x0(%r9,%rax,4),%rsi
0x131D: nop (%rax)
```

bubbleSort:B41 *loop_1: 1/3*

```
0x1320: movq (%rdx),%xmm0
0x1324: pshufd %xmm0,%xmm5,%xmm2
0x1329: movd %xmm0,%cx
0x132D: movd %xmm2,%ax
0x1331: pshufd %xmm0,%xmm1,%xmm1
0x1336: cmp %eax,%ecx
0x1338: jle 0x6(%rip) bubbleSort:B43
```

bubbleSort:B42 *loop_1: 2/3*

```
0x133A: movq %xmm1, (%rdx)
```

bubbleSort:B43 *loop_1: 3/3*

```
0x133E: add $0x4,%rdx
0x1342: cmp %rsi,%rdx
0x1345: jnz 0xfffffddb(%rip) bubbleSort:B41
```

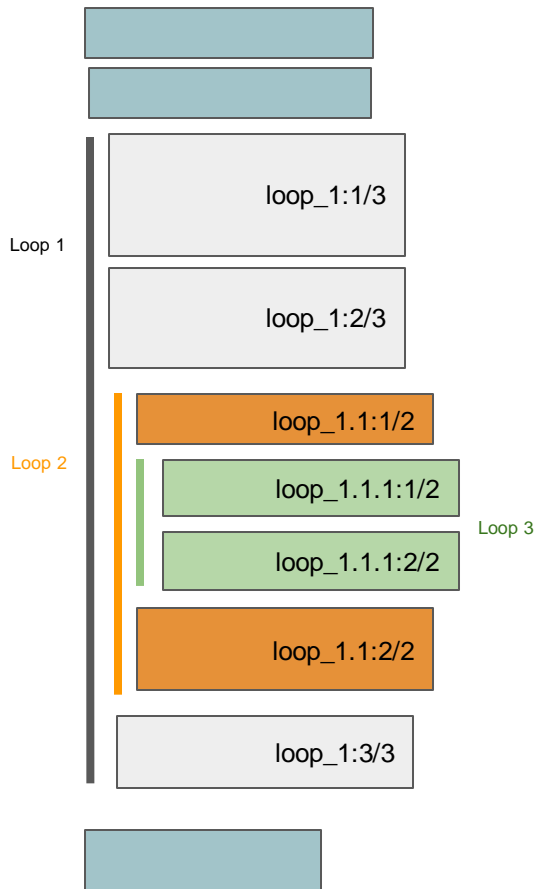
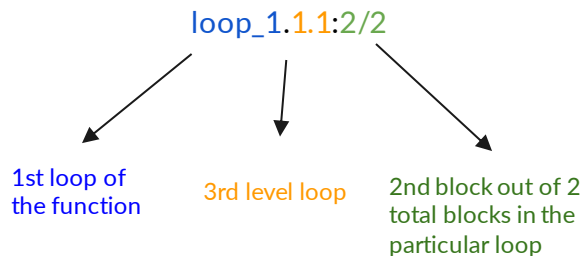
bubbleSort:B44 *loop_1: 2/2*

```
0x1347: mov %r8,%eax
0x134A: test %r8,%r8
0x134D: jnz 0xffffffc3(%rip) bubbleSort:B40
```

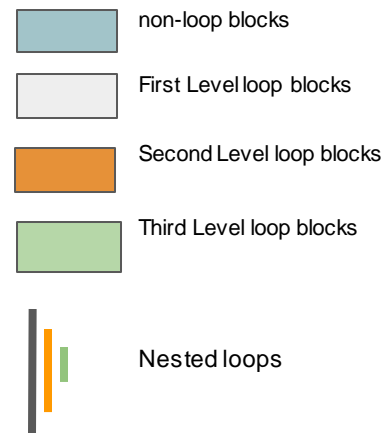
Ideal Loop Constructions

Blocks ordered by Virtual Memory Address

- **Indentation:** Nested Loop blocks
- **Numbering loop blocks:**
Eg. Block with indication loop_1.1.1:2/2



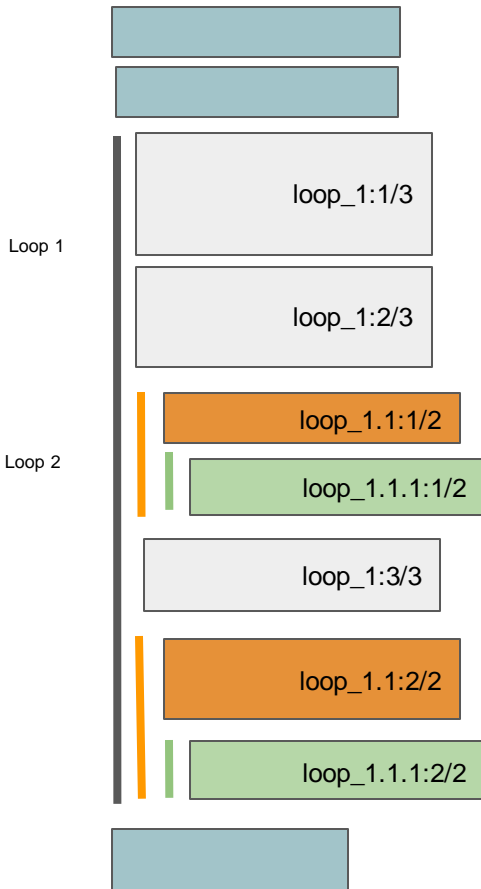
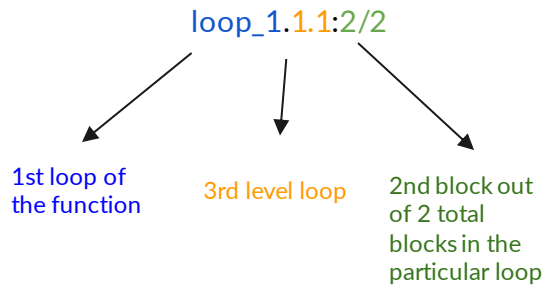
Colors used for clarity



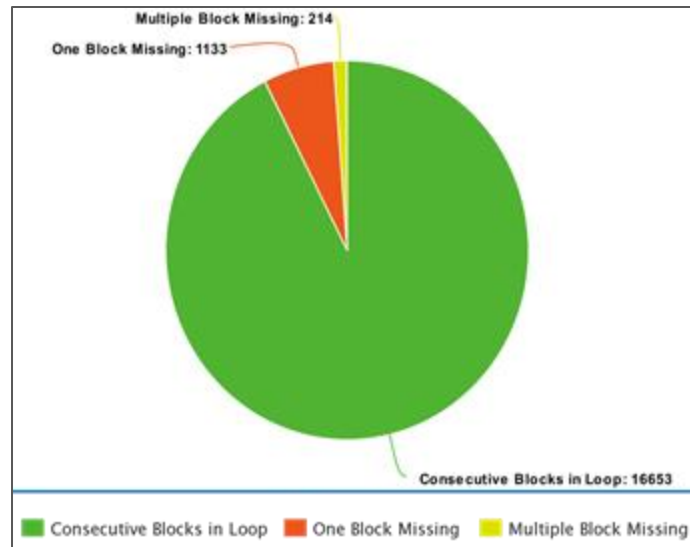
Real Case for Loops

Blocks ordered by Virtual Memory Address

Blocks in a particular loop are not ordered sequentially



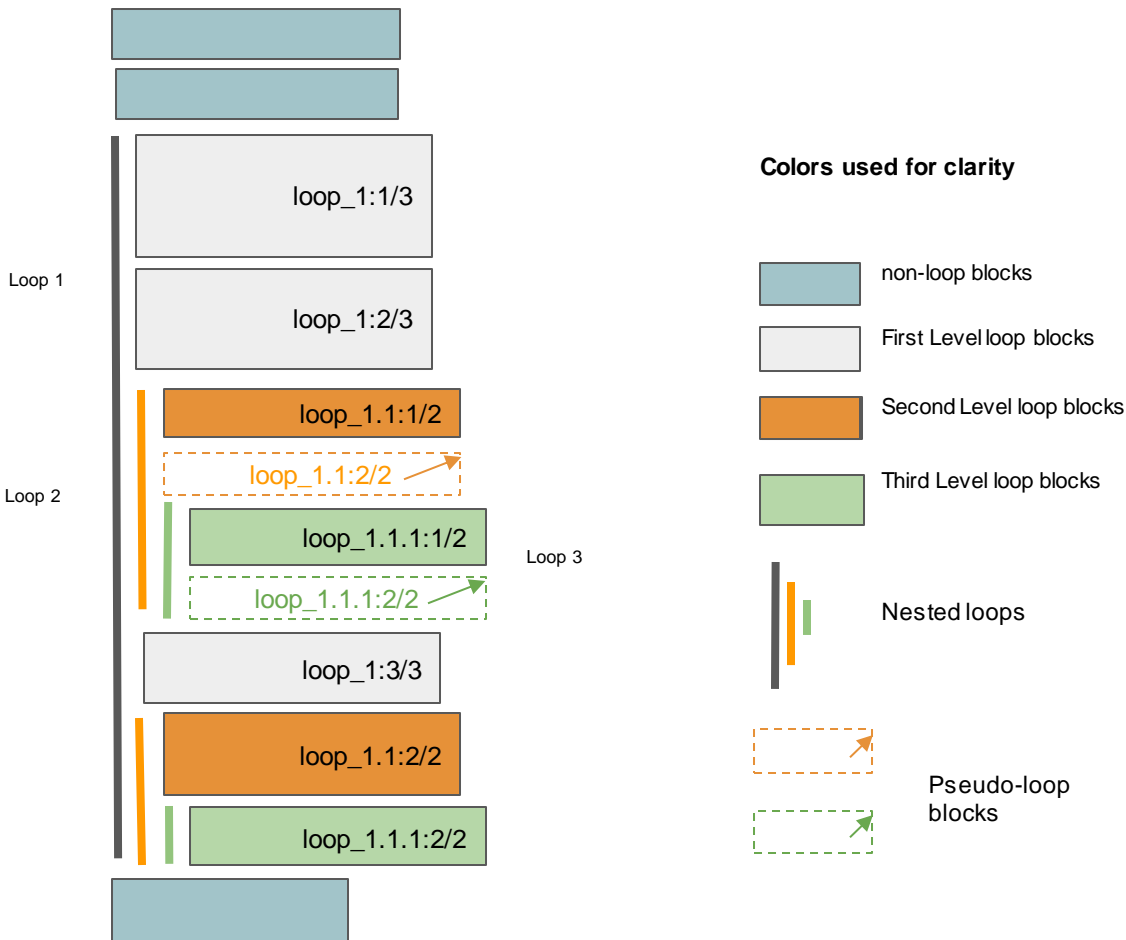
RAJA Performance Suite



Pseudo Loop Blocks Proposal

Loop blocks have different orders.

- Hard to understand which blocks are in single loop.
- To **avoid confusion** **Pseudo-loop blocks** are introduced.
- Dotted blocks arrange the instruction blocks of a particular loop together.



Pseudo Loop Blocks

In Memory Address order,
the pseudo loop blocks are
inserted to maintain logical
order of blocks inside a loop

The image displays a debugger interface with two main panes: 'Source: LTIMES-OMP.cpp' on the left and 'Disassembly View: 1' on the right. The source code shows a function `void ltimes::runOpenMPVariant` with nested loops. The disassembly view shows the corresponding assembly instructions, with pseudo loop blocks highlighted in yellow and labeled with addresses and loop names.

Source: LTIMES-OMP.cpp

```
20
21 void ltimes::runOpenMPVariant(Varintid vid, size_t RAJAPERS_UMU
22 {
23     #if defined(RAJA_ENABLE_OPENMP) && defined(RUN_OPENMP)
24
25     const Index_type run_reps = getRunReps();
26
27     LTIMES_DATA_SETUP;
28
29     switch ( vid ) {
30
31     case Base_OpenMP : {
32
33         startTimer();
34         for (RepIndex_type irep = 0; irep < run_reps; ++irep) {
35
36             #pragma omp parallel for
37             for (Index_type z = 0; z < num_z; ++z) {
38                 for (Index_type g = 0; g < num_g; ++g) {
39                     for (Index_type m = 0; m < num_m; ++m) {
40                         for (Index_type d = 0; d < num_d; ++d) {
41                             LTIMES_BODY;
42                         }
43                     }
44                 }
45             }
46         }
47         stopTimer();
48         break;
49     }
50
51     case Lambda_OpenMP : {
52
53         auto ltimes_base_lam = [=](Index_type d, Index_type z,
54                                 Index_type g, Index_type m) {
55             LTIMES_BODY;
56         };
57
58         startTimer();
59         for (RepIndex_type irep = 0; irep < run_reps; ++irep) {
60
61             #pragma omp parallel for
62             for (Index_type z = 0; z < num_z; ++z) {
63                 for (Index_type g = 0; g < num_g; ++g) {
64                     for (Index_type m = 0; m < num_m; ++m) {
65                         for (Index_type d = 0; d < num_d; ++d) {
66                             ltimes_base_lam(d, z, g, m);
67                         }
68                     }
69                 }
70             }
71         }
72     }
73
74 }
```

Disassembly View: 1

Order By: Memory Address

Annotations:

- When pseudo blocks are clicked (Orange arrow pointing to the pseudo block for `loop_1.1.1/3`).
- Highlights the original block (Purple arrow pointing to the original block for `loop_1.1.1/3`).

Load more

Order of Disassembly View

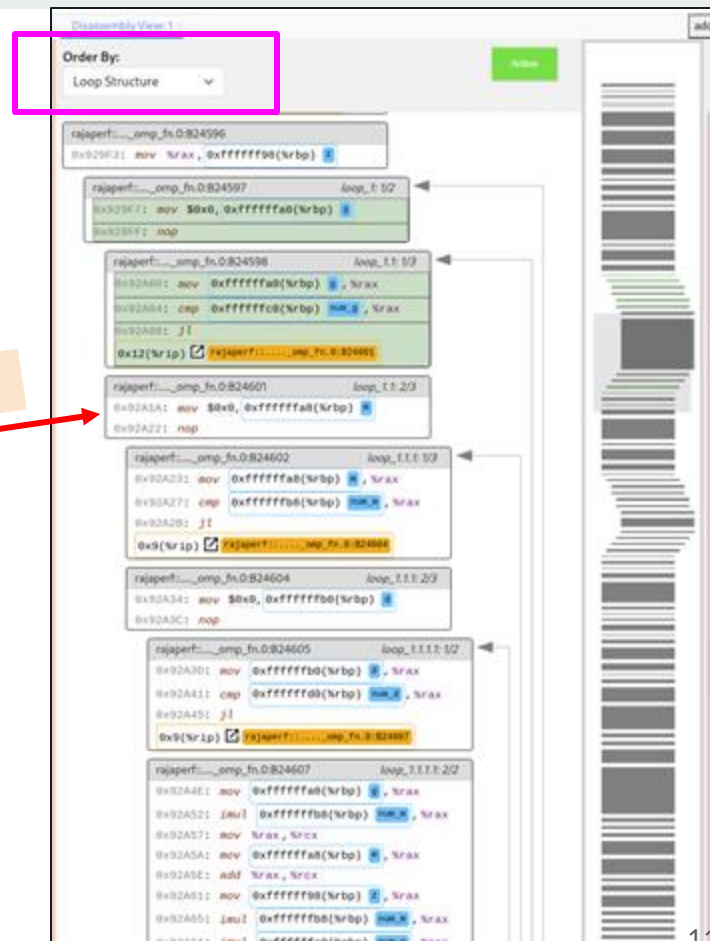
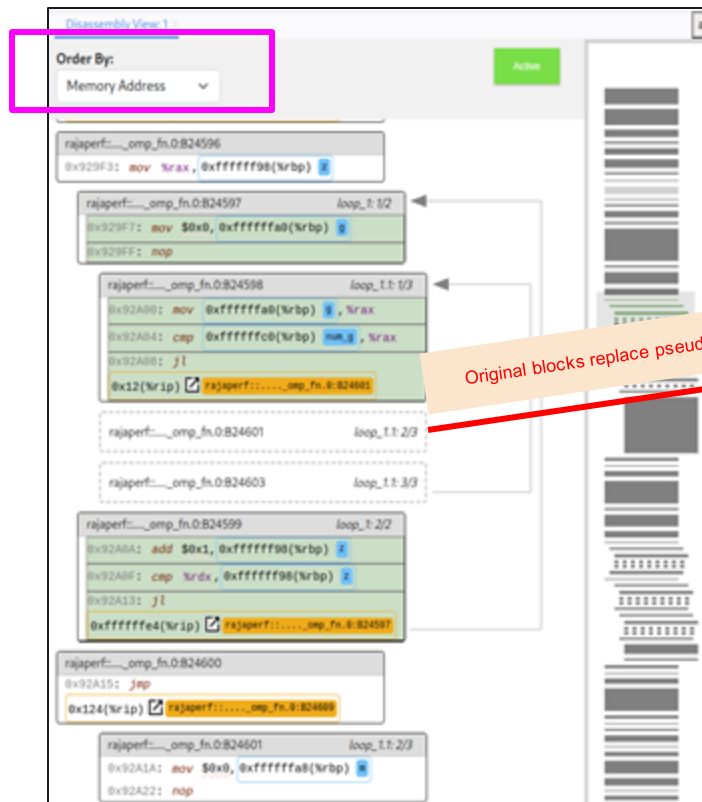
Loop blocks visualized in 2 approaches:

1. Memory Address

Order of instructions according to memory address

2. Loop Structure

- Order of instructions according to loop structure.
- Reordering blocks out of address order



Loop Backedges

Becomes difficult to visualize at a glance

Backedges intersect with each other when using original blocks



To avoid difficulty, pseudo loop blocks used instead of original blocks.

Minimap Overview

Disassembly View: 1 X

Order By: Memory Address Active

@x929ED: *jnl*
@x14c(%rip) ↗ rajaperf::.....omp_fn.0:B24612

rajaperf::.....omp_fn.0:B24599
@x929F3: *mov* %rax, 0xffffffff98(%rbp) z

rajaperf::.....omp_fn.0:B24600 *loop_1: 1/2*
@x929F7: *mov* \$0x0, 0xffffffffa0(%rbp) 0
@x929FF: *nop*

rajaperf::.....omp_fn.0:B24601 *loop_1.1: 1/3*
@x92A00: *mov* 0xfffffffffa0(%rbp) 0, %rax
@x92A04: *cmp* 0xffffffffc0(%rbp) num_q, %rax
@x92A08: *j1*
@x12(%rip) ↗ rajaperf::.....omp_fn.0:B24604

rajaperf::.....omp_fn.0:B24604 *loop_1.1: 2/3*

rajaperf::.....omp_fn.0:B24606 *loop_1.1: 3/3*

rajaperf::.....omp_fn.0:B24602 *loop_1: 2/2*
@x92A0A: *add* \$0x1, 0xffffffff98(%rbp) z
@x92A0F: *cmp* %rdx, 0xffffffff98(%rbp) z
@x92A13: *j1*
@xffffffffe4(%rip) ↗ rajaperf::.....omp_fn.0:B24608

Minimap created to get the disassembly overview

Each rectangle height - no. of instructions in blocks

Light Grey Lines - System defined functions

Dark Grey Lines - User defined functions

Dotted Lines- Pseudo loop Blocks

Indentation- Nested loops

Brush- Current Location

Multiple Disassembly Views

Multiple disassembly views
with different colors

For visualizing multiple source
code at once

For example-

Dis View 1 - Green

Dis View 2 - Yellow

Dis View 3 - Orange and so on

The screenshot displays a disassembly tool interface. On the left, the 'Source Files' pane shows a project structure for 'root/RAJA-PERFSUITE-Deb/RAJAPerf'. The 'src' directory contains subdirectories like 'common', 'algorithm', and 'apps', each with various source files. The main area on the right, titled 'Disassembly View: 1', shows assembly code for different sections. The code is color-coded: green for the first section, yellow for the second, and orange for the third. The first section, labeled '_init:B0', contains instructions like 'endbr64', 'sub \$0x8, %rsp', 'mov 0x20cf01(%rip), %rax', 'test %rax, %rax', and 'jz 0x4(%rip)'. The second section, labeled '_init:B1', contains 'call %rax'. The third section, labeled '_init:B2', contains 'add \$0x8, %rsp' and 'ret near (%rsp)'. Below these, there are sections for 'target870:B3', '_Znam:B4', '_ZNKSt7_c...capacityEv:B5', and '_ZNSt7_cx...EEaSERKS4_B6', each with their respective assembly instructions. A vertical scrollbar is visible on the right side of the disassembly view.

Bidirectional & Many to many relation

In addition with Dyninst,
CcNav corresponds source and
disassembly code in both ways.

The screenshot displays the CcNav tool interface, which facilitates a bidirectional relationship between source code and disassembly code. The interface is divided into two main panes: 'Source View' on the left and 'Disassembly View' on the right.

Source View (Left Pane): Shows the source code of the file 'LTIMES-OMP.cpp'. The code includes headers, namespace definitions, and a function 'void LTIMES::runOpenMPVariant'. The code is wrapped in a 'Wrapping' block. The function 'runOpenMPVariant' is defined, taking 'VariantID vid' and 'size_t RAJAPERF_UN' as arguments. It includes a switch statement for different OpenMP variants, with a case for 'Base_OpenMP' that contains a nested loop structure for parallel execution.

Disassembly View (Right Pane): Shows the disassembly code corresponding to the source code. The disassembly is organized into sections, each with a label and a list of instructions. The sections are:

- _initB71069:** Contains instructions for setting up the initial state, including 'endbrq', 'sub \$0x8, %rsp', 'mov 0x20cfd5(%rip), %rax', 'test %rax, %rax', and 'jz 0x4(%rip)'.
- _initB71070:** Contains the instruction 'call %rax'.
- _initB71071:** Contains instructions for setting up the initial state, including 'add \$0x8, %rsp' and 'ret near (%rip)'.
- targetB7080:** Contains instructions for setting up the target state, including 'endbrq' and 'REPZ jmp 0x20c6f5(%rip)'.
- _ZnamB1:** Contains instructions for setting up the target state, including 'endbrq' and 'REPZ jmp 0x20c2bd(%rip)'.
- _ZNKSA7_c.capacityEvB2:** Contains instructions for setting up the target state, including 'endbrq' and 'REPZ jmp 0x20c2b5(%rip)'.
- _ZNKSA7_c.capacityEvB3:** Contains instructions for setting up the target state, including 'endbrq' and 'REPZ jmp 0x20c2ad(%rip)'.

The disassembly view also includes a 'Order By' dropdown menu set to 'Memory Address' and a 'Filter' button.

Other features

Description of the operands are shown in the tool tip

Arrows used to represent which lines have correspondence

The screenshot shows a debugger window with assembly code. A tooltip is displayed over the instruction at address 0x92A23: `mov 0xffffffff, 0x9(%rip)`. The tooltip contains the following information:

- Instruction: MOV
- Meaning: Move
- Notes: copies data from one location to another, (1) r/m = r; (2) r = r/m;
- Opcode: 0xA0...0xA3

Arrows indicate the correspondence between the assembly code and the tooltip.

```
4 int main(int argc, char *argv[])
5 {
6     int i, j;
7     for (i = 0; i < 10000; i++)
8         for (j = 0; j < 10000; j += argc)
9             {
10                if (unlikely(i * j == 900))
11                {
12                    printf("Hello %d %d %d", i, j, argc);
13                }
14            }
15     return 42;
16 }
```

No
correspondence
line in dis view

0x1082: `nop 0x0(%rax,%rax,1)`

main:B6 loop_1: 1/3

```
0x1088: mov %r13d,%ebp
0x108B: mov %r12d,%ebx
0x108E: cmp $0x270f,%r12d
0x1095: jnl 0x21(%rip) main:810
```

When clicked it
Jumps to
defined block

main:B7 loop_1: 2/3

```
0x1097: nop 0x0(%rax,%rax,1)
```

main:B8 loop_1: 1/4

```
0x10A0: cmp $0x384,%ebp
```

Other features (continued)

Dyninst gives information regarding variables which are renamed in the disassembly code

The screenshot displays a mapping between C code and its disassembly. In the C code (lines 62-76), variables `z`, `g`, `m`, and `d` are used in nested loops. Red boxes highlight these variables. An arrow points from the `d` in line 67 to the disassembly windows on the right. The disassembly windows show instructions where these variables are mapped to specific registers or memory locations, with red boxes highlighting the mappings: `0xffffffffb8(%rbp)` for `z`, `0xffffffffc8(%rbp)` for `g`, and `0xffffffffe8(%rbp)` for `m`. The `d` variable is mapped to `%rax` and `num_d`.

The screenshot shows a disassembly window for a function named `rajaperf::apps::LTIMES::runOpenMPVariant_omp_fn.0: B24599`. The function name is truncated to a fixed length. The disassembly instructions are: `0x92A0A: add $0x1, 0xffffffff98(%rbp) z`, `0x92A0F: cmp %rdx, 0xffffffff98(%rbp) z`, `0x92A13: jl`, and `0xffffffffe4(%rip) [rajaperf::..._omp_fn.0: B24597]`. A red box highlights the function name, and a blue box highlights the variable `z`.

The function names are also truncated to a fixed length



Disassembly

Control Flow Graph



©

18

Thank you!

Main Goal: To improve the human side of compilation analysis with visualization

The screenshot displays a software development environment with three main panels:

- Source View:** Shows C code for a function `void LTIMES::runOpenMP(...) const`. It includes nested loops for `Index_type` and `Index_type g`, with a `switch` statement for `Index_type`. The code is annotated with `rappt::_jump` and `rappt::_jump` labels.
- Assembly View 1:** Shows the assembly code for the function, with instructions like `mov $0x0, %eax` and `rappt::_jump` labels. It includes a `Order By: Memory Address` dropdown.
- Assembly View 2:** Shows the assembly code for the function, with instructions like `mov $0x0, %eax` and `rappt::_jump` labels. It includes a `Order By: Memory Address` dropdown.

The control flow graph (CFG) is visible in the background, showing the flow of execution between different blocks of code.

Name: Shadmaan Hye
Email: praptishadmaan@gmail.com
Github: <https://github.com/Prapti-044/dis-viz.git>