CcNav: Understanding Compiler Optimizations in Binary Code Supplemental Materials

Sabin Devkota, Pascal Aschwanden, Adam Kunen, Matthew Legendre, and Katherine E. Isaacs

1 Full Hierarchical Task Analysis

We include the full task hierarchy derived from our task analysis. Yet undefined terms are described below.

Goal: Understand performance / Identify optimizations

- T1 Understand/Identify compiled structure
  - T1.1 Match source code with binary code
  - T1.2 Identify/Relate structures with code
    - T1.2.1 Identify loops
    - T1.2.2 Identify functions
  - T1.3 Annotate relations
    - T1.3.1 Annotate registers with variables
    - T1.3.2 Annotate loops & loop internal structure
  - T1.4 Trace variable
- T2 Understand optimizations
  - T2.1 Find areas of interest
    - T2.1.1 Overview of binary code
    - T2.1.2 Winnow code
      - T2.1.2.1 Winnow to specific loop
      - T2.1.2.2 Winnow to function
      - T2.1.2.3 Winnow to line of code
      - T2.1.2.4 Winnow based on performance metric
    - T2.1.3 Identify anomalous code
  - T2.2 Identify optimizations
    - T2.2.1 Identify inlining
    - T2.2.2 Identify vectorization
    - T2.2.3 Identify code hoisting
    - T2.2.4 Identify loop unrolling
  - T2.3 Assess optimizations
    - T2.3.1 Assess amount of optimization present
    - T2.3.2 Relate to performance metrics
  - T2.4 Compare generated code
    - T2.4.1 Compare code with different optimizations
    - T2.4.2 Compare code with different source
    - T2.4.3 Compare code with different compilers
  - T2.5 Annotate optimizations

Loop internal structure refers to instructions related to the loop body (what performs the computation) and the preamble and postamble (which manage the iteration).

Code hoisting is another optimization which moves a computation out of its enclosing loop when the computation is unnecessary to repeat. We did not have a real example of expected code hoisting, so we did not prioritize this optimization.

Anomalous code is ill-defined. Presently it is described as "I'll know it when I see it."

Performance metrics can be real or simulated measures of actual performance. We expect this will require extending our automated analysis. It is not addressed by this paper.

2 Basic Evaluation Tasks Completed by P0 and P1

Our evaluation sessions with participants P0 and P1 included several basic evaluation tasks. We decided to not repeat those in the sessions with P2 and P3 because (1) P0 and P1 had completed them easily and (2) we wanted to afford more time to the tasks that were closer to a real analysis session. We list the basic tasks completed by P0 and P1 here:

- Find a specific line in the source code.
- Given the line of source code, find the function that contains it.
- Given a function, what functions are inlined inside of it?
- Given a loop, what function calls are made in it?

2.1 Extended Evaluation Task Descriptions

We provide our detailed observations regarding the pair analytics actions of our participants below.

E1: Identify the assembly of a loop containing a selected line of source code. Because a loop spans multiple lines and the mapping between source code and disassembly is imperfect, this task has an implication beyond straightforward highlighting. All participants started by asking to click on the first line of the loop, highlighting the corresponding code, and continued their analysis without pause.

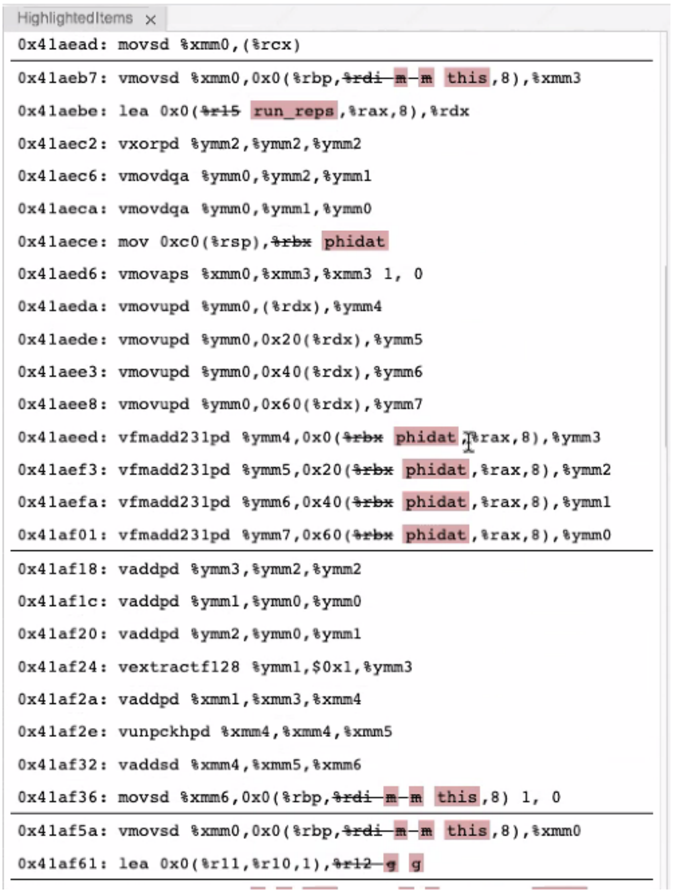
P0, P1, and P2 next examined the loop hierarchy view. P0 noted the source code line is the top of a quadrupally nested loop which was not fully depicted in the loop hierarchy. The facilitator clarified that the source code-to-disassembly mapping only maps the clicked line of the loop and not the whole body. P0 asked to click on the top level loop shown in the loop hierarchy. This selected the whole loop body in the source and showed the complete nesting in the hierarchy.

P1 guessed the correct loop by looking at the partial loop hierarchy, reasoning, "Loop 3 must be the outer loop, so 3.1 must be the one we're on." To verify, they asked to click on Loop 3.1 and noted the one-to-one correspondence with the source code loops. P2, on the other hand, asked to perform a range search by dragging and selecting the whole loop body in the source code. They immediately noted the complete loop hierarchy in the hierarchy view. P2 also verified by asking to click on loop 3.1 and observing the same line highlighted in the source code.

P3 looked at the selected disassembly directly and found the index variable 'z' annotated, matching the loop source code. When asked for the loop name in the loop hierarchy, they asked to click on the top level loop `loop3`. Observing that both source code and loop hierarchy have five levels of nested loops, P3 guessed the correct loop.

E2: Identify/Assess vectorization in that loop. P1, P2, and P3 all noted they did not recall exact vector instructions, but communicated they would look for them. P0 required some background knowledge on vectorization and the facilitator instructed that the presence of one of the vector registers would indicate vectorization. P1 and P2 were suggested names of vector registers.

- *Sabin Devkota and Katherine E. Isaacs are with University of Arizona. E-mail: {devkotasabin@email.arizona.edu | kisaacs@cs.arizona.edu}.*
- *Pascal Aschwanden, Adam Kunen, and Matthew Legendre are with LLNL. E-mail: {aschwanden1 | kunen1 | legendre1}@llnl.gov.*



```
HighlightedItems ×
0x41aead: movsd %xmm0,(%rcx)
0x41aeb7: vmovsd %xmm0,0x0(%rbp,~~%rdi m m~~ this,8),%xmm3
0x41aebe: lea 0x0(~~%r15~~ run_reps,%rax,8),%rdx
0x41aec2: vxorpd %ymm2,%ymm2,%ymm2
0x41aec6: vmovdqa %ymm0,%ymm2,%ymm1
0x41aeca: vmovdqa %ymm0,%ymm1,%ymm0
0x41aece: mov 0xc0(%rsp),~~%rbx~~ phidat
0x41aed6: vmovaps %xmm0,%xmm3,%xmm3 1, 0
0x41aeda: vmovupd %ymm0,(%rdx),%ymm4
0x41aede: vmovupd %ymm0,0x20(%rdx),%ymm5
0x41aee3: vmovupd %ymm0,0x40(%rdx),%ymm6
0x41aee8: vmovupd %ymm0,0x60(%rdx),%ymm7
0x41aeed: vfmadd231pd %ymm4,0x0(~~%rbx~~ phidat,%rax,8),%ymm3
0x41aef3: vfmadd231pd %ymm5,0x20(~~%rbx~~ phidat,%rax,8),%ymm2
0x41aefa: vfmadd231pd %ymm6,0x40(~~%rbx~~ phidat,%rax,8),%ymm1
0x41af01: vfmadd231pd %ymm7,0x60(~~%rbx~~ phidat,%rax,8),%ymm0
0x41af18: vaddpd %ymm3,%ymm2,%ymm2
0x41af1c: vaddpd %ymm1,%ymm0,%ymm0
0x41af20: vaddpd %ymm2,%ymm0,%ymm1
0x41af24: vextractf128 %ymm1,$0x1,%ymm3
0x41af2a: vaddpd %xmm1,%xmm3,%xmm4
0x41af2e: vunpckhpd %xmm4,%xmm4,%xmm5
0x41af32: vaddsd %xmm4,%xmm5,%xmm6
0x41af36: movsd %xmm6,0x0(%rbp,~~%rdi m m~~ this,8) 1, 0
0x41af5a: vmovsd %xmm0,0x0(%rbp,~~%rdi m m~~ this,8),%xmm0
0x41af61: lea 0x0(%r11,%r10,1),~~%r12 g~~ g
```

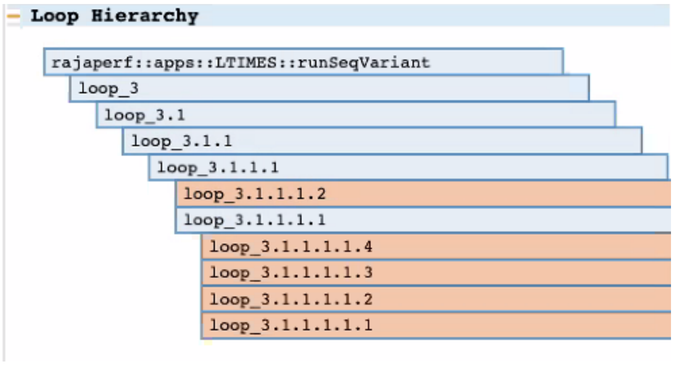
Fig. 1. This screenshot captures the selected disassembly in the Highlighted Items View. Evaluation participant P3 recognized the highlighted `phidat` variable to verify their position. They then discovered the `vfmadd231pd` instructions indicative of vectorization.

P0, P1, and P2 all started by asking to click on loop 3.1 in the hierarchy. P0 asked to scroll through the instructions in the selected items view. They found an instruction using a vector register and then turned back to the disassembly view to click on that instruction. They concluded the loop has vectorization after verifying the instruction links backs to the starting line of source code.

P1 and P2 expressed interest in searching. The facilitator reminded P1 that `ctrl-f` could be used. P2 remembered. Both asked to search for vector registers in the selected items view, found corresponding vector instructions (`vfmadd231pd`) from the search, and concluded the loop was vectorized.

P3 took a different strategy from the other participants. They asked to click on the body of the innermost loop in the source code, saying they planned to look for the arithmetic instructions and possible unrolling therein. Scanning through the selected items view (Fig. 1), they remarked the renamed variable annotations are helpful for identifying the data loading instructions. After some scrolling, they found four fused multiply-add instructions (`vfmadd231pd`). They said, "That's amazing actually. It does look like its vectorized, but it's doing multiple of them back to back, so it's highly unrolled, so it's vectorized really well."

E3: Compare/Assess multiple variants in the source code. The LTIMES application has several versions of the same computation. In this task, we focused on two: a) a "base-sequential" ("Base") version with nested four loops, and b) a "RAJA-sequential" ("RAJA") version where loops are implemented using RAJA constructs and thus the quadruple nesting is not explicitly written in the source file. Some participants also chose to look at a third variant, "lambda-sequential"



```
− Loop Hierarchy
rajaperf::apps::LTIMES::runSeqVariant
  loop_3
    loop_3.1
      loop_3.1.1
        loop_3.1.1.1
          loop_3.1.1.1.2
          loop_3.1.1.1.1
            loop_3.1.1.1.1.4
            loop_3.1.1.1.1.3
            loop_3.1.1.1.1.2
            loop_3.1.1.1.1.1
```

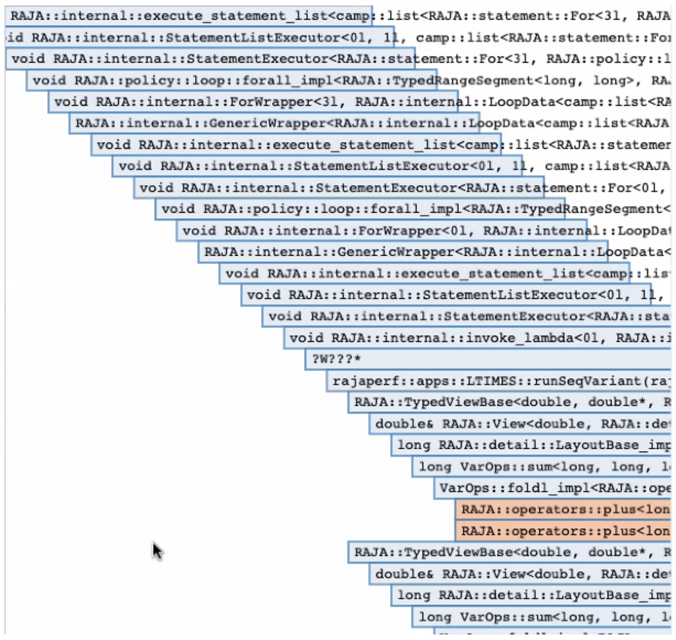
Fig. 2. Loop hierarchy view. Evaluation participant P1 determines the leaves are four variants of the same loop, generated by the compiler to aid loop unrolling.

("Lambda") which is like Base, but uses a lambda function for the body. This task was free-form by design and each participant approached it with a different strategy.

P0 looked at the RAJA version in the source code, observing there were only two lines not grayed out. They asked to click on the first line and then looked at the disassembly in the selected items view. P0 then turned their attention to the CFG view, examining the function names in the nodes. Next they examined the loop hierarchy view and asked to click on the top level. A new line was highlighted in the source code: a lambda function. The function inlining tree refreshed with several nodes as well, so P0 asked to collapse the view. P0 asked why the original line and the lambda function were both highlighted. After a reminder that loop hierarchy selection is by full loop, P0 investigated the source code view for any other highlighted lines. They correctly hypothesized that the highlighted disassembly was then showing the loop body, but said they were not sure how to assess the differences further due to lack of experience in this kind of analysis.

P1 asked to click on the top level function in the loop hierarchy, which they surmised would contain all versions. They then asked to collapse the function inlining tree since it contained a lot of items. They asked to click on a specific loop in the loop hierarchy. They recognized this loop was associated with the RAJA version, but wanted to check the Lambda version first. They then asked to click on the top-level loop in the Lambda version in the source code. P1 remarked the top-level loops in both Base and Lambda looked similar. They then directed the facilitator to navigate down the loop hierarchy, asking to click on specific loops for further comparison. P1 said the second level loops look similar and hypothesized the optimizations are only in the inner two loops. At the first innermost loop among the four leaf nodes (Fig. 2), P1 hypothesized that the inner loops in both versions are vectorized and that the leaf loops are "fixing up the ends for the vectorization unroll." They repeat the process with the Base version, confirming their expectation.

P1 then asked to click on the source line with RAJA construct. They noted this does not result in a loop in the loop hierarchy view. They turned to the CFG view, needing to scroll. They mentioned the CFG is not helpful because of lack of instructions in the basic blocks. P1 asked to click again on the RAJA construct in the source code to get back to the previous state. They then explored the function inlining view, recalling it had "kernel stuff" from previous exploration (Fig. 3, full context: Fig. 4,). P1 asked to scroll through the inlining tree. They recognized a function from their previous experience with RAJA kernels and asked to click on it. They observed that the loop hierarchy view has changed and decided to explore further. P1 asked to click on `loop2.1.1`. The loop hierarchy view updated to show more nesting. P1 identified the quadruple-nested loop that was the target of their search. They remarked the code structure is similar to the base version, but obfuscated by the long call stack. They further identified candidates for loop preamble and postamble instructions in the CFG View (Fig. 5, full context: (Fig. 6)).



```
RAJA::internal::execute_statement_list<camp::list<RAJA::statement::For<3l, RAJA
id RAJA::internal::StatementListExecutor<0l, 1l, camp::list<RAJA::statement::For
void RAJA::internal::StatementExecutor<RAJA::statement::For<3l, RAJA::policy::l
  void RAJA::policy::loop::forall_impl<RAJA::TypedRangeSegment<long, long>, RA
    void RAJA::internal::ForWrapper<3l, RAJA::internal::LoopData<camp::list<RA
      RAJA::internal::GenericWrapper<RAJA::internal::LoopData<camp::list<RAJA
        void RAJA::internal::execute_statement_list<camp::list<RAJA::statemer
          void RAJA::internal::StatementListExecutor<0l, 1l, camp::list<RAJA
            void RAJA::internal::StatementExecutor<RAJA::statement::For<0l,
              void RAJA::policy::loop::forall_impl<RAJA::TypedRangeSegment<
                void RAJA::internal::ForWrapper<0l, RAJA::internal::LoopDat
                  RAJA::internal::GenericWrapper<RAJA::internal::LoopData<
                    void RAJA::internal::execute_statement_list<camp::lis
                      void RAJA::internal::StatementListExecutor<0l, 1l,
                        void RAJA::internal::StatementExecutor<RAJA::sta
                          void RAJA::internal::invoke_lambda<0l, RAJA::i
                            ?W???*
                              rajaperf::apps::LTIMES::runSeqVariant(ra
                                RAJA::TypedViewBase<double, double*, R
                                  double& RAJA::View<double, RAJA::de
                                    long RAJA::detail::LayoutBase_imp
                                      long VarOps::sum<long, long, l
                                        VarOps::foldl_impl<RAJA::ope
                                          RAJA::operators::plus<lon
                                          RAJA::operators::plus<lon
                                RAJA::TypedViewBase<double, double*, R
                                  double& RAJA::View<double, RAJA::de
                                    long RAJA::detail::LayoutBase_imp
                                      long VarOps::sum<long, long, l
```

Fig. 3. This screenshot is a zoomed in version of Evaluation participant P1's area of interest in the kernel code. The full interface is shown in Fig. 4.

P2 said they wanted to further examine the base version first. They asked to click on the top-level loop in this version and then to scroll through the selected items view for an overview of instructions. They also examined the CFG view, but expressed confusion at the disconnected nodes. They then moved to the call graph view and reasoned the disconnected nodes in the CFG were due to a data setup line in the source. P2 then asked to range-select the entire base version. They examined the call graph view further but determined it was not helpful and instead asked to browse the selected items view.

Next, P2 asked to click on the RAJA construct in the RAJA version. Noticing no loops in the loop hierarchy, they then asked to click on the `for` loop which repeats the loop kernel multiple times. They examined the loop hierarchy, saw one loop (`loop2`), and noted there were only a few disassembly lines selected. They remarked they could tell it was making an indirect call from the selected disassembly. Examining the source code further, P2 noted a lambda function was called in the RAJA construct, hypothesizing it was the indirect call. They asked to find the source code of that lambda function and click on it. They noted the disassembly selected by this operation is what they sought. They asked to scroll through the selected items view and remarked on vectorization present in this version as well.

P2 then returned to the loop hierarchy view and asked to click on the top level loop (`loop2`). Noticing more loops showing up in the hierarchy, they asked to click on levels beneath it. They directed the facilitator to perform clicks between the source and loop hierarchies to repeat the actions for the base version for comparison. P2 then repeated their strategy of going through the lambda function to return to the RAJA view. P2 hypothesized that both versions have everything inlined, but there is more overhead in the RAJA version for the indirect call. They qualified their finding, noting their RAJA knowledge is not too deep. (Their findings are consistent with performance data not used in the evaluation.)

P3 started by asking to click on the top-level `for` loop in the Lambda version. P3 expressed confusion that the loop hierarchy did not show the inner loops. They did not recall the option to click the loop. P3 then asked to click on the source line with the innermost `for` loop. P3 observed the same loop unrolling structure they previously found in the Base version. They wanted to click on the loop body but it had no mappings to the disassembly. P3 then asked to scroll through the selected items. Spotting the annotations in the disassembly for variable `phidat`, P3 hypothesized they were looking at the data setup. P3 said they were looking for the arithmetic instructions of the loop body. They switched to the full disassembly view after noting that the disassembly in the selected items view was not enough because the source line only maps to the loop setup in disassembly. P3 then found some non-highlighted arithmetic instructions and said "that's completely what we want to see." P3 remarked "highlighted terms is really tempting but sometimes you just really have to look." From these instructions, P3 concluded that this variant was vectorized like the Base.

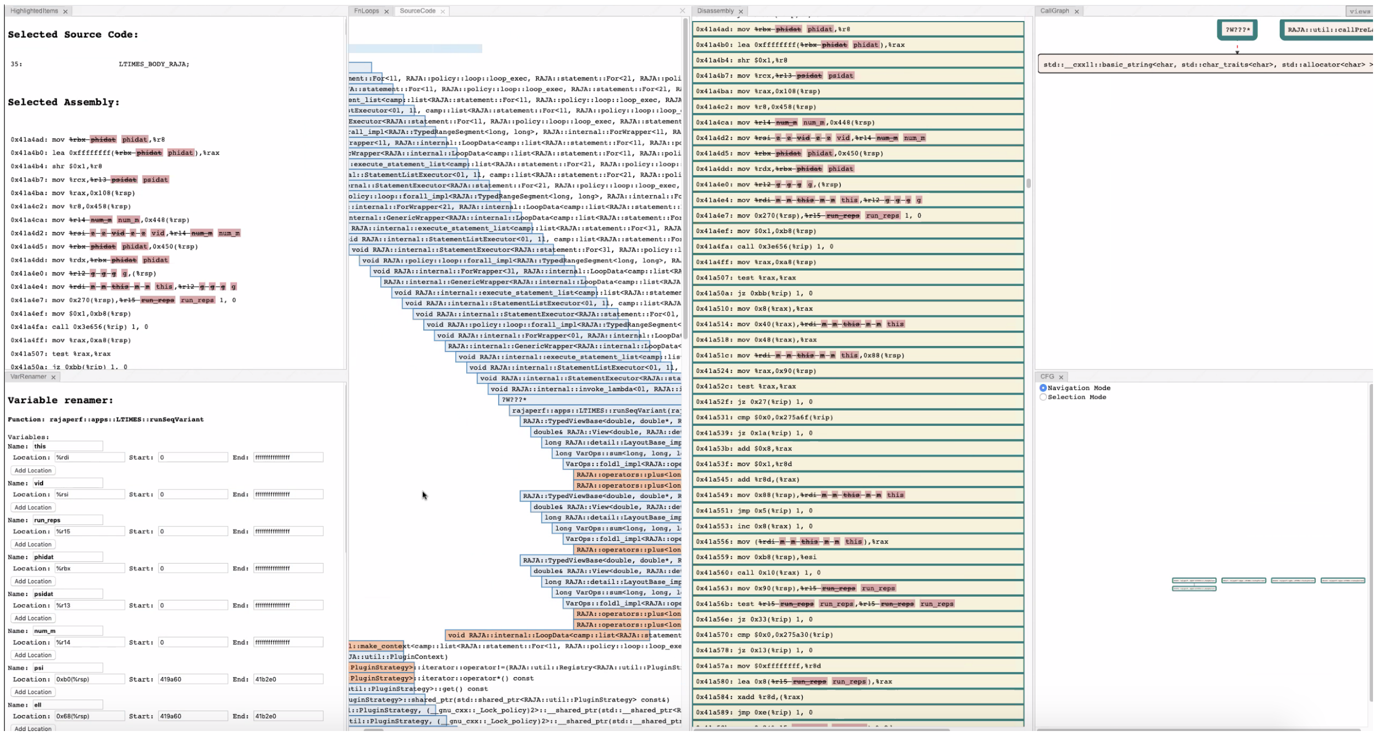
P3 then asked to click on the RAJA construct in the source view, which highlighted few instructions in the disassembly. After a pause, the facilitator suggested exploring the loop hierarchy. However, P3 continued with the source code view and asked to click on the enclosing `for` loop. This updated the loop hierarchy to show `loop2`. P3 expressed wanting to drill down the hierarchy but did not recall the option to click on the loop. P3 instead asked to click on the RAJA construct again and started exploring the CFG, suggesting it might contain the loop body. In this case, the k-hop filter did not show a loop. They asked to click on some of the nodes, but did not find the loops. P3 remarked the CFG was too low-level without the loop information and there was not enough context. They then asked to click on the same line of source code to go back to the previous state. They examined the text inside the highlighted basic blocks in CFG. P3 hypothesized the current selections to be part of a branch and following the path downward would find the start of the loop. Their remarks seemed to indicate confusion about what the CFG was showing.

3 Early Prototype Figures

We include images of other early prototypes. Specifically, we include a second pre-CFG prototype (Fig. 7), the complete version of the matching prototype from the paper text (Fig. 8), and an example of a prototype with full instructions in the CFG nodes, similar to CFGExplorer (Fig. 9) with its subsequent change to smaller nodes (Fig. 10). Fig. 9 shows the CFG nodes can be very large in terms of number of instructions, distorting the graph topology.

4 Acknowledgements

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-812737.



Selected Source Code:

```
35:                     LTIMES_BODY_RAJA;
```

Selected Assembly:

```
0x41a4ad: mov ~~%rbx phidat~~ phidat,%r8
0x41a4b0: lea 0xffffffff(~~%rbx phidat~~ phidat),%rax
0x41a4b4: shr $0x1,%r8
0x41a4b7: mov %rcx,~~%r13 psidat~~ psidat
0x41a4ba: mov %rax,0x108(%rsp)
0x41a4c2: mov %r8,0x458(%rsp)
0x41a4ca: mov ~~%r14 num_m~~ num_m,0x448(%rsp)
0x41a4d2: mov ~~%rsi z z vid z z~~ vid,~~%r14 num_m~~ num_m
0x41a4d5: mov ~~%rbx phidat~~ phidat,0x450(%rsp)
0x41a4dd: mov %rdx,~~%rbx phidat~~ phidat
0x41a4e0: mov ~~%r12 g g g~~ g,(%rsp)
0x41a4e4: mov ~~%rdi m m this m m~~ this,~~%r12 g g g~~ g
0x41a4e7: mov 0x270(%rsp),~~%r15 run_reps~~ run_reps 1, 0
0x41a4ef: mov $0x1,0xb8(%rsp)
0x41a4fa: call 0x3e656(%rip) 1, 0
0x41a4ff: mov %rax,0xa8(%rsp)
0x41a507: test %rax,%rax
0x41a50a: jz 0xbb(%rip) 1, 0
```

Variable renamer:

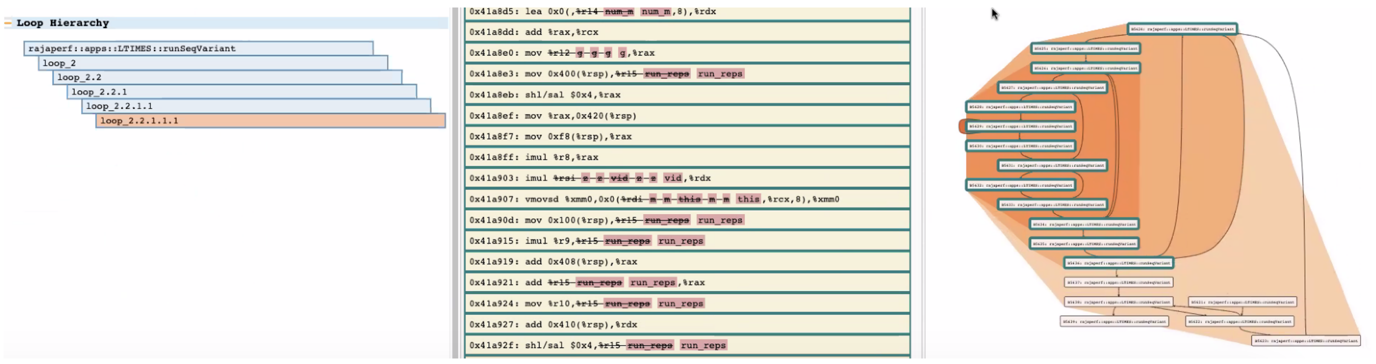
Function: rajaperf::apps::LTIMES::runSeqVariant

Variables:

Name: this			
Location: %rdi	Start: 0	End: ffffffffffffffff	
Name: vid			
Location: %rsi	Start: 0	End: ffffffffffffffff	
Name: run_reps			
Location: %r15	Start: 0	End: ffffffffffffffff	
Name: phidat			
Location: %rbx	Start: 0	End: ffffffffffffffff	
Name: psidat			
Location: %r13	Start: 0	End: ffffffffffffffff	
Name: num_m			
Location: %r14	Start: 0	End: ffffffffffffffff	
Name: psi			
Location: 0xb0(%rsp)	Start: 419a60	End: 41b2e0	
Name: ell			
Location: 0x68(%rsp)	Start: 419a60	End: 41b2e0	

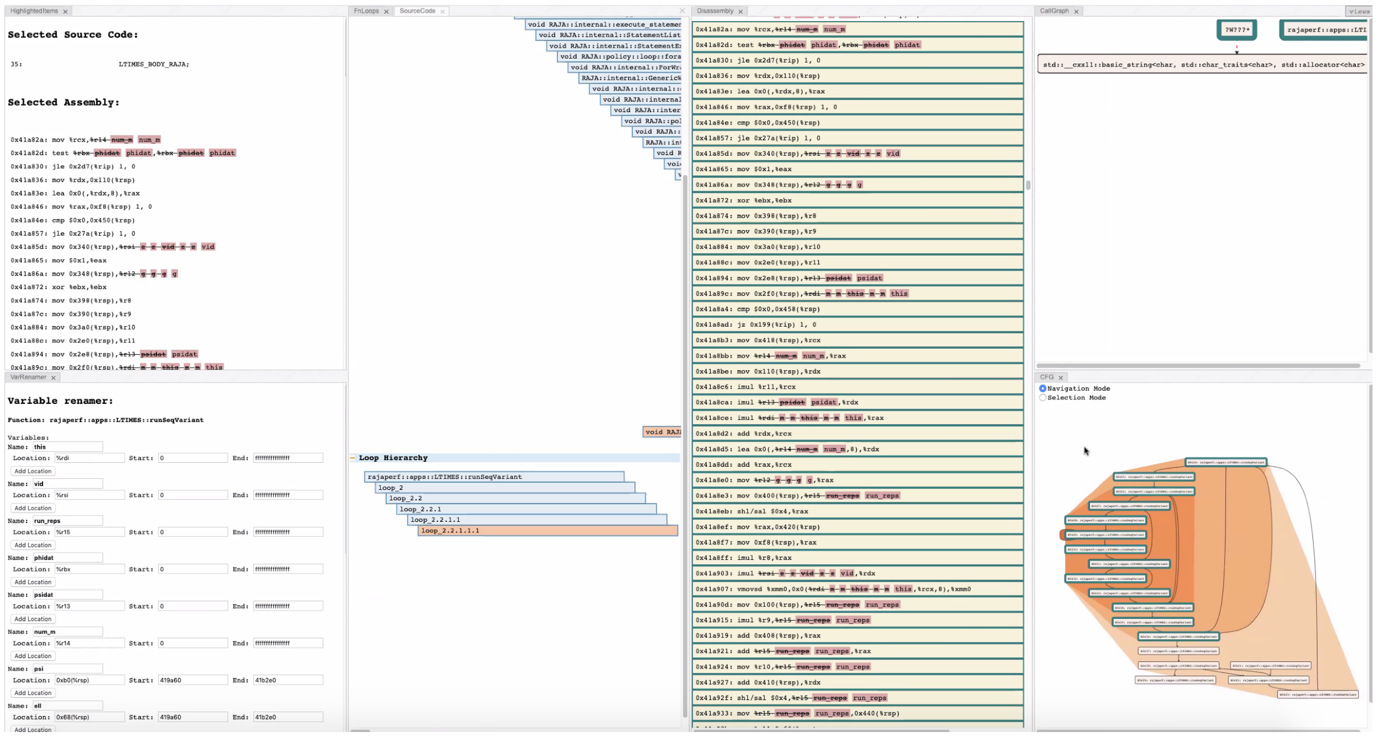
```
0x41a4ad: mov ~~%rbx phidat~~ phidat,%r8
0x41a4b0: lea 0xffffffff(~~%rbx phidat~~ phidat),%rax
0x41a4b4: shr $0x1,%r8
0x41a4b7: mov %rcx,~~%r13 psidat~~ psidat
0x41a4ba: mov %rax,0x108(%rsp)
0x41a4c2: mov %r8,0x458(%rsp)
0x41a4ca: mov ~~%r14 num_m~~ num_m,0x448(%rsp)
0x41a4d2: mov ~~%rsi z z vid z z~~ vid,~~%r14 num_m~~ num_m
0x41a4d5: mov ~~%rbx phidat~~ phidat,0x450(%rsp)
0x41a4dd: mov %rdx,~~%rbx phidat~~ phidat
0x41a4e0: mov ~~%r12 g g g~~ g,(%rsp)
0x41a4e4: mov ~~%rdi m m this m m~~ this,~~%r12 g g g~~ g
0x41a4e7: mov 0x270(%rsp),~~%r15 run_reps~~ run_reps 1, 0
0x41a4ef: mov $0x1,0xb8(%rsp)
0x41a4fa: call 0x3e656(%rip) 1, 0
0x41a4ff: mov %rax,0xa8(%rsp)
0x41a507: test %rax,%rax
0x41a50a: jz 0xbb(%rip) 1, 0
0x41a510: mov 0x8(%rax),%rax
0x41a514: mov 0x40(%rax),~~%rdi m m this m m~~ this
0x41a518: mov 0x48(%rax),%rax
0x41a51c: mov ~~%rdi m m this m m~~ this,0x88(%rsp)
0x41a524: mov %rax,0x90(%rsp)
0x41a52c: test %rax,%rax
0x41a52f: jz 0x27(%rip) 1, 0
0x41a531: cmp $0x0,0x275a6f(%rip)
0x41a539: jz 0x1a(%rip) 1, 0
0x41a53b: add $0x8,%rax
0x41a53f: mov $0x1,%r8d
0x41a545: add %r8d,(%rax)
0x41a549: mov 0x88(%rsp),~~%rdi m m this m m~~ this
0x41a551: jmp 0x5(%rip) 1, 0
0x41a553: inc 0x8(%rax) 1, 0
0x41a556: mov (~~%rdi m m this m m~~ this),%rax
0x41a559: mov 0xb8(%rsp),%esi
0x41a560: call 0x10(%rax) 1, 0
0x41a563: mov 0x90(%rsp),~~%r15 run_reps~~ run_reps
0x41a56b: test ~~%r15 run_reps~~ run_reps,~~%r15 run_reps~~ run_reps
0x41a56e: jz 0x33(%rip) 1, 0
0x41a570: cmp $0x0,0x275a30(%rip)
0x41a578: jz 0x13(%rip) 1, 0
0x41a57a: mov $0xffffffff,%r8d
0x41a580: lea 0x8(~~%r15 run_reps~~ run_reps),%rax
0x41a584: xadd %r8d,(%rax)
0x41a589: jmp 0xe(%rip) 1, 0
```

Fig. 4. This screenshot shows a window into the Function Inlining Tree as directed by Evaluation participant P1. In this view, they had asked to stack the Source Code View so they could focus more on the other views. They recognized this particularly deep inlining chain as an indicator of kernel code and looked for recognizable functions. This is also an example of a disconnected CFG.



```
0x41a8d5: lea 0x0(,~~%r14 num_m~~ num_m,8),%rdx
0x41a8dd: add %rax,%rcx
0x41a8e0: mov ~~%r12 g g g~~ g,%rax
0x41a8e3: mov 0x400(%rsp),~~%r15 run_reps~~ run_reps
0x41a8eb: shl/sal $0x4,%rax
0x41a8ef: mov %rax,0x420(%rsp)
0x41a8f7: mov 0xf8(%rsp),%rax
0x41a8ff: imul %r8,%rax
0x41a903: imul ~~%rsi z z vid z z~~ vid,%rdx
0x41a907: vmovsd %xmm0,0x0(~~%rdi m m this m m~~ this,%rcx,8),%xmm0
0x41a90d: mov 0x100(%rsp),~~%r15 run_reps~~ run_reps
0x41a915: imul %r9,~~%r15 run_reps~~ run_reps
0x41a919: add 0x408(%rsp),%rax
0x41a921: add ~~%r15 run_reps~~ run_reps,%rax
0x41a924: mov %r10,~~%r15 run_reps~~ run_reps
0x41a927: add 0x410(%rsp),%rdx
0x41a92f: shl/sal $0x4,~~%r15 run_reps~~ run_reps
```

Fig. 5. Drilling down into the loop hierarchy reveals nested loops in the CFG subgraph.



Selected Source Code:

```
35:                    LTIMES_BODY_RAJA;
```

Selected Assembly:

```
0x41a82a: mov %rcx,~~%r14 num_m~~ num_m
0x41a82d: test ~~%rbx phidat~~ phidat,~~%rbx phidat~~ phidat
0x41a830: jle 0x2d7(%rip) 1, 0
0x41a836: mov %rdx,0x110(%rsp)
0x41a83e: lea 0x0(,%rdx,8),%rax
0x41a846: mov %rax,0xf8(%rsp) 1, 0
0x41a84e: cmp $0x0,0x450(%rsp)
0x41a857: jle 0x27a(%rip) 1, 0
0x41a85d: mov 0x340(%rsp),~~%rsi z z vid z z~~ vid
0x41a865: mov $0x1,%eax
0x41a86a: mov 0x348(%rsp),~~%r12 g g g~~ g
0x41a872: xor %ebx,%ebx
0x41a874: mov 0x398(%rsp),%r8
0x41a87c: mov 0x390(%rsp),%r9
0x41a884: mov 0x3a0(%rsp),%r10
0x41a88c: mov 0x2e0(%rsp),%r11
0x41a894: mov 0x2e8(%rsp),~~%r13 psidat~~ psidat
0x41a89c: mov 0x2f0(%rsp),~~%rdi m m this m m~~ this
```

Variable renamer:

Function: rajaperf::apps::LTIMES::runSeqVariant

Variables:

Name	Location	Start	End
this	%rdi	0	ffffffffffffffff
vid	%rsi	0	ffffffffffffffff
run_reps	%r15	0	ffffffffffffffff
phidat	%rbx	0	ffffffffffffffff
psidat	%r13	0	ffffffffffffffff
num_m	%r14	0	ffffffffffffffff
psi	0xb0(%rsp)	419a60	41b2e0
ell	0x68(%rsp)	419a60	41b2e0

Add Location

```
void RAJA::internal::execute_statemer
 void RAJA::internal::StatementList
  void RAJA::internal::StatementE
   void RAJA::policy::loop::fora
    void RAJA::internal::ForWr
     RAJA::internal::GenericW
      void RAJA::internal::
       void RAJA::internal
        void RAJA::inter
         void RAJA::pol
          void RAJA::
           RAJA::int
            void R
             voi
              v
void RAJ
```

Loop Hierarchy

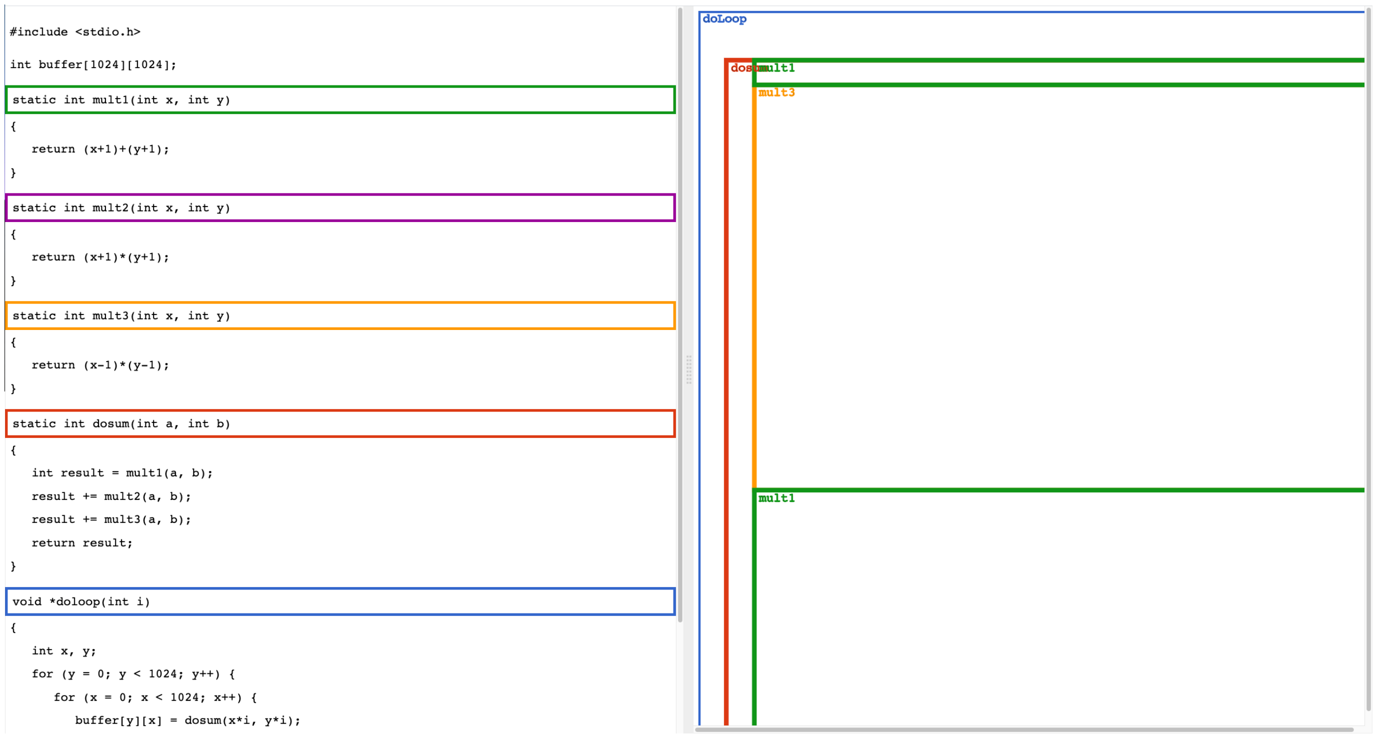
```
rajaperf::apps::LTIMES::runSeqVariant
 loop_2
  loop_2.2
   loop_2.2.1
    loop_2.2.1.1
     loop_2.2.1.1.1
```

```
0x41a82a: mov %rcx,~~%r14 num_m~~ num_m
0x41a82d: test ~~%rbx phidat~~ phidat,~~%rbx phidat~~ phidat
0x41a830: jle 0x2d7(%rip) 1, 0
0x41a836: mov %rdx,0x110(%rsp)
0x41a83e: lea 0x0(,%rdx,8),%rax
0x41a846: mov %rax,0xf8(%rsp) 1, 0
0x41a84e: cmp $0x0,0x450(%rsp)
0x41a857: jle 0x27a(%rip) 1, 0
0x41a85d: mov 0x340(%rsp),~~%rsi z z vid z z~~ vid
0x41a865: mov $0x1,%eax
0x41a86a: mov 0x348(%rsp),~~%r12 g g g~~ g
0x41a872: xor %ebx,%ebx
0x41a874: mov 0x398(%rsp),%r8
0x41a87c: mov 0x390(%rsp),%r9
0x41a884: mov 0x3a0(%rsp),%r10
0x41a88c: mov 0x2e0(%rsp),%r11
0x41a894: mov 0x2e8(%rsp),~~%r13 psidat~~ psidat
0x41a89c: mov 0x2f0(%rsp),~~%rdi m m this m m~~ this
0x41a8a4: cmp $0x0,0x458(%rsp)
0x41a8ad: jz 0x199(%rip) 1, 0
0x41a8b3: mov 0x418(%rsp),%rcx
0x41a8bb: mov ~~%r14 num_m~~ num_m,%rax
0x41a8be: mov 0x110(%rsp),%rdx
0x41a8c6: imul %r11,%rcx
0x41a8ca: imul ~~%r13 psidat~~ psidat,%rdx
0x41a8ce: imul ~~%rdi m m this m m~~ this,%rax
0x41a8d2: add %rdx,%rcx
0x41a8d5: lea 0x0(,~~%r14 num_m~~ num_m,8),%rdx
0x41a8dd: add %rax,%rcx
0x41a8e0: mov ~~%r12 g g g~~ g,%rax
0x41a8e3: mov 0x400(%rsp),~~%r15 run_reps~~ run_reps
0x41a8eb: shl/sal $0x4,%rax
0x41a8ef: mov %rax,0x420(%rsp)
0x41a8f7: mov 0xf8(%rsp),%rax
0x41a8ff: imul %r8,%rax
0x41a903: imul ~~%rsi z z vid z z~~ vid,%rdx
0x41a907: vmovsd %xmm0,0x0(~~%rdi m m this m m~~ this,%rcx,8),%xmm0
0x41a90d: mov 0x100(%rsp),~~%r15 run_reps~~ run_reps
0x41a915: imul %r9,~~%r15 run_reps~~ run_reps
0x41a919: add 0x408(%rsp),%rax
0x41a921: add ~~%r15 run_reps~~ run_reps,%rax
0x41a924: mov %r10,~~%r15 run_reps~~ run_reps
0x41a927: add 0x410(%rsp),%rdx
0x41a92f: shl/sal $0x4,~~%r15 run_reps~~ run_reps
0x41a933: mov ~~%r15 run_reps~~ run_reps,0x440(%rsp)
```

```
?W77?*    rajaperf::apps::LTI
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
```

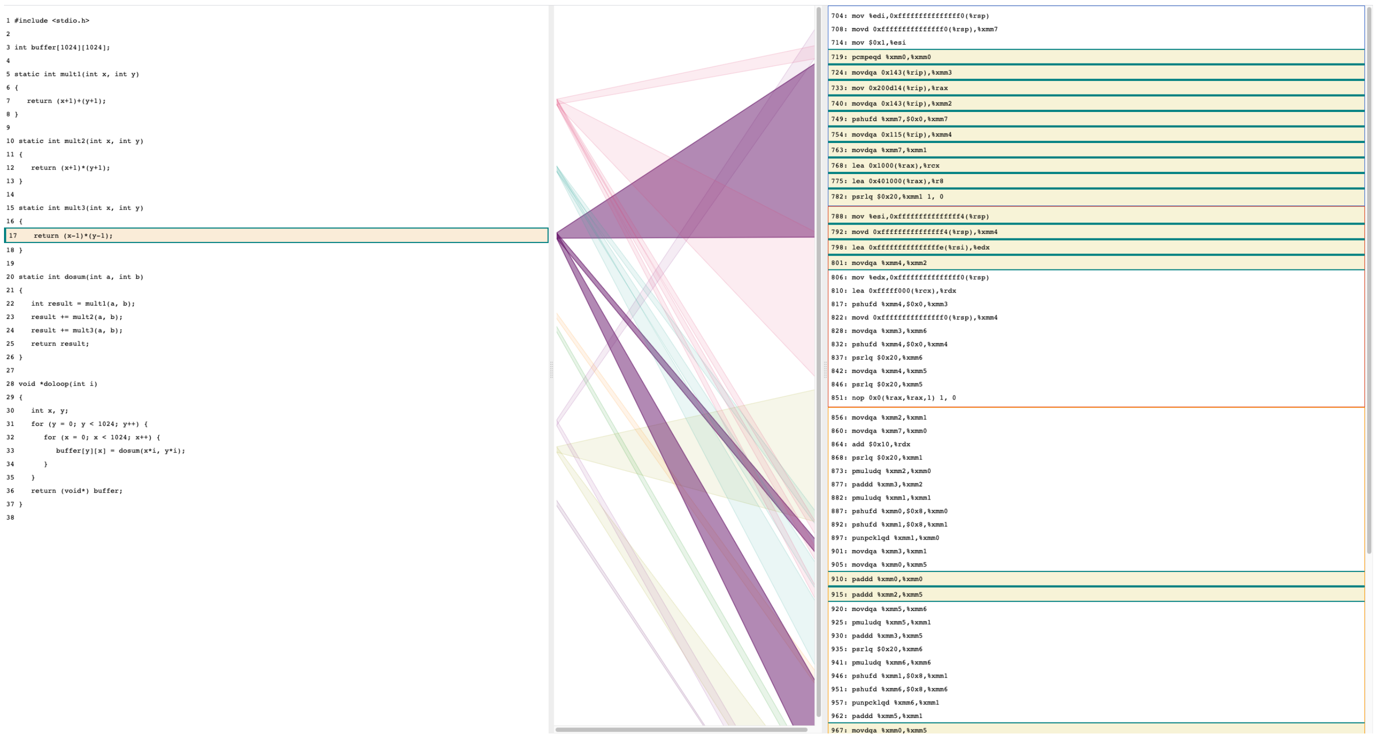
Navigation Mode [✓]
Selection Mode

Fig. 6. This is a screenshot from the evaluation with Evaluation participant P1. After using the Function Inlining Tree, they used the Loop Hierarchy View and retrieved the nested loops shown in the CFG View. A cropped version of this figure is shown in the paper.



```
#include <stdio.h>

int buffer[1024][1024];

static int mult1(int x, int y)
{
   return (x+1)+(y+1);
}

static int mult2(int x, int y)
{
   return (x+1)*(y+1);
}

static int mult3(int x, int y)
{
   return (x-1)*(y-1);
}

static int dosum(int a, int b)
{
   int result = mult1(a, b);
   result += mult2(a, b);
   result += mult3(a, b);
   return result;
}

void *doloop(int i)
{
   int x, y;
   for (y = 0; y < 1024; y++) {
      for (x = 0; x < 1024; x++)
         buffer[y][x] = dosum(x*i, y*i);
```

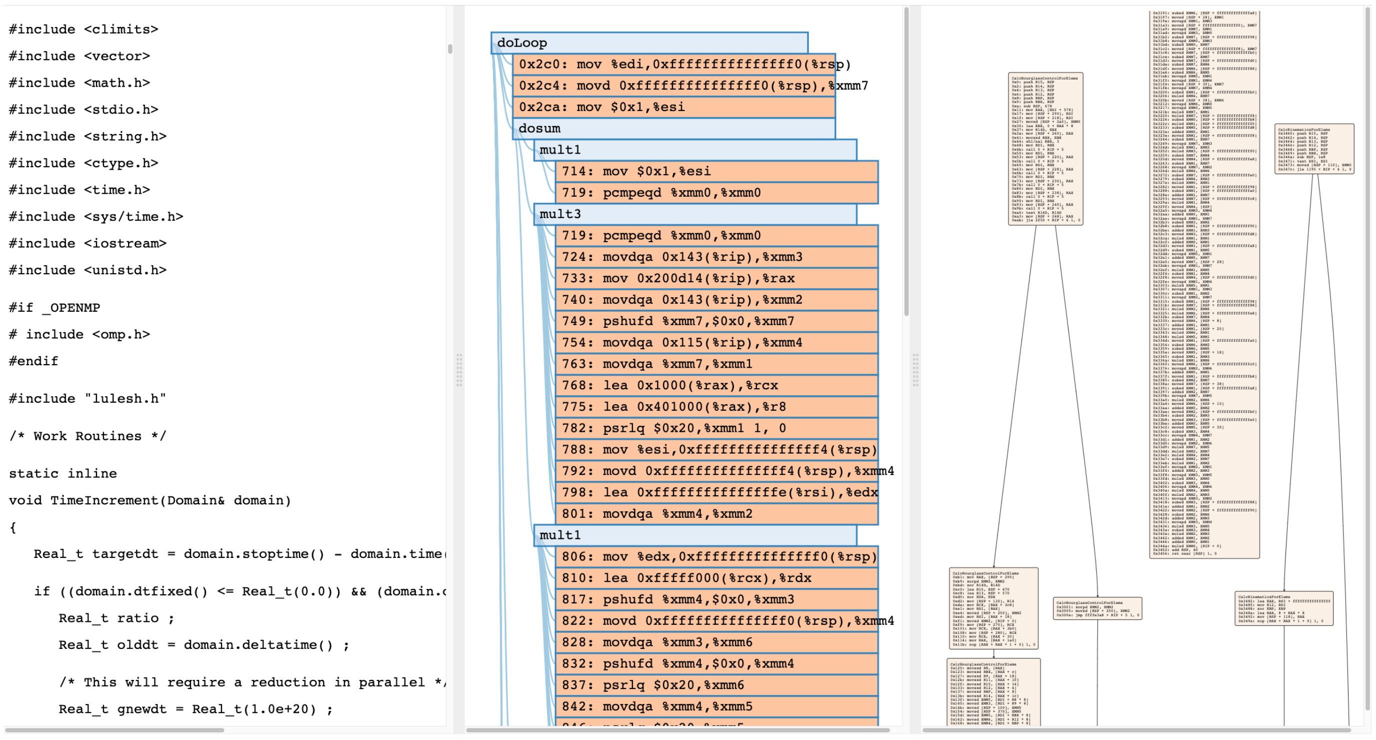
Fig. 7. This early prototype did not have a CFG View. Instead it uses color outlining to show a correspondence between source code and inlining derived from the disassembly.



```
#include <stdio.h>

int buffer[1024][1024];

static int mult1(int x, int y)
{
   return (x+1)+(y+1);
}

static int mult2(int x, int y)
{
   return (x+1)*(y+1);
}

static int mult3(int x, int y)
{
   return (x-1)*(y-1);
}

static int dosum(int a, int b)
{
   int result = mult1(a, b);
   result += mult2(a, b);
   result += mult3(a, b);
   return result;
}

void *doloop(int i)
{
   int x, y;
   for (y = 0; y < 1024; y++) {
      for (x = 0; x < 1024; x++) {
         buffer[y][x] = dosum(x*i, y*i);
      }
   }
   return (void*) buffer;
}
```

```
704: mov %edi,0xfffffffffffffff0(%rsp)
708: movd 0xfffffffffffffff0(%rsp),%xmm7
714: mov $0x1,%esi
719: pcmpeqd %xmm0,%xmm0
724: movdqa 0x143(%rip),%xmm3
733: mov 0x200d14(%rip),%rax
740: movdqa 0x143(%rip),%xmm2
749: pshufd %xmm7,$0x0,%xmm7
754: movdqa 0x115(%rip),%xmm4
763: movdqa %xmm7,%xmm1
768: lea 0x1000(%rax),%rcx
775: lea 0x401000(%rax),%r8
782: psrlq $0x20,%xmm1 1, 0
788: mov %esi,0xfffffffffffffff4(%rsp)
792: movd 0xfffffffffffffff4(%rsp),%xmm4
798: lea 0xfffffffffffffffe(%rsi),%edx
801: movdqa %xmm4,%xmm2
806: mov %edx,0xfffffffffffffff0(%rsp)
810: lea 0xfffff000(%rcx),%rdx
817: pshufd %xmm4,$0x0,%xmm3
822: movd 0xfffffffffffffff0(%rsp),%xmm4
828: movdqa %xmm3,%xmm6
832: pshufd %xmm4,$0x0,%xmm4
837: psrlq $0x20,%xmm6
842: movdqa %xmm4,%xmm5
846: psrlq $0x20,%xmm5
851: nop 0x0(%rax,%rax,1) 1, 0
856: movdqa %xmm2,%xmm1
860: movdqa %xmm7,%xmm0
864: add $0x10,%rdx
868: psrlq $0x20,%xmm1
873: pmuludq %xmm2,%xmm0
877: paddd %xmm3,%xmm2
882: pmuludq %xmm1,%xmm1
887: pshufd %xmm0,$0x8,%xmm0
892: pshufd %xmm1,$0x8,%xmm1
897: punpcklqd %xmm1,%xmm0
901: movdqa %xmm3,%xmm1
905: movdqa %xmm0,%xmm5
910: paddd %xmm0,%xmm0
915: paddd %xmm2,%xmm5
920: movdqa %xmm5,%xmm6
925: pmuludq %xmm5,%xmm1
930: paddd %xmm3,%xmm5
935: psrlq $0x20,%xmm6
941: pmuludq %xmm6,%xmm6
946: pshufd %xmm1,$0x8,%xmm1
951: pshufd %xmm6,$0x8,%xmm6
957: punpcklqd %xmm6,%xmm1
962: paddd %xmm5,%xmm1
967: movdqa %xmm0,%xmm5
```

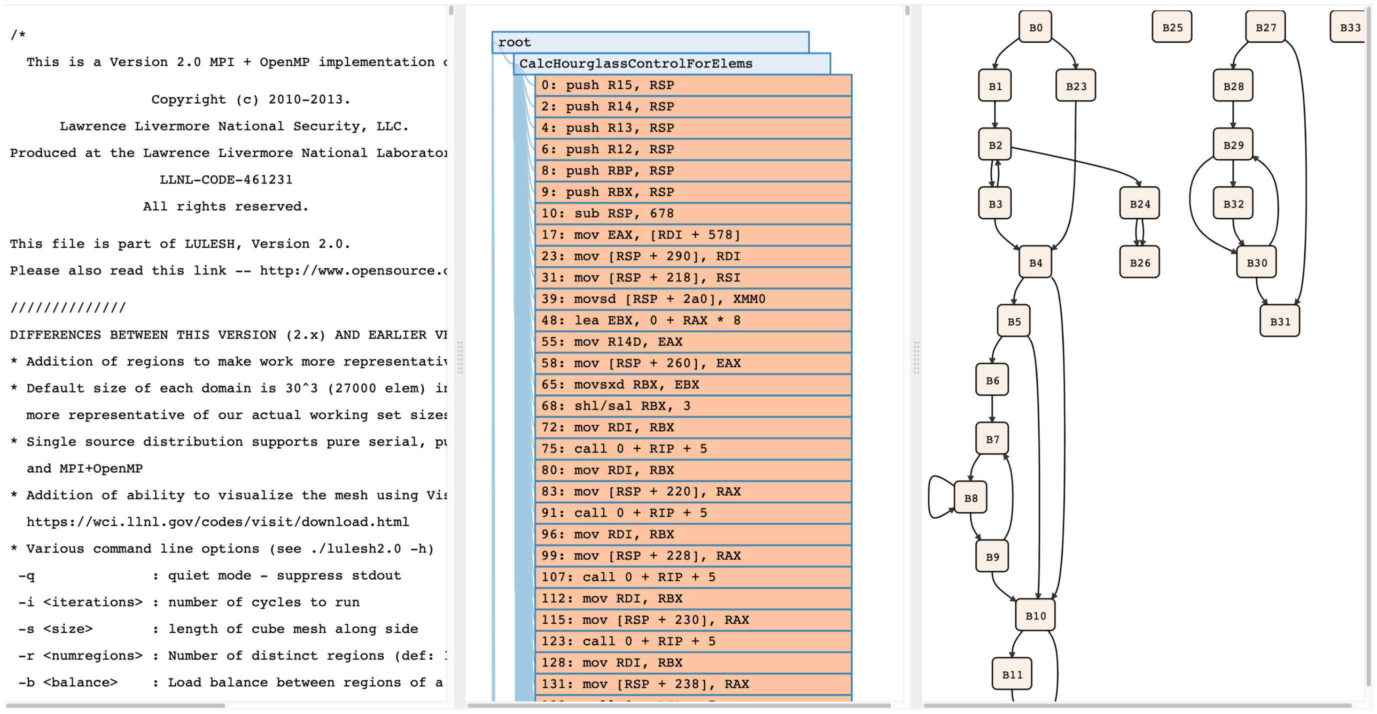
Fig. 8. This is the full screenshot of the source code to disassembly matching prototype shown in the paper.



```
#include <climits>
#include <vector>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <sys/time.h>
#include <iostream>
#include <unistd.h>

#if _OPENMP
# include <omp.h>
#endif

#include "lulesh.h"

/* Work Routines */

static inline
void TimeIncrement(Domain& domain)
{
   Real_t targetdt = domain.stoptime() - domain.time(
   if ((domain.dtfixed() <= Real_t(0.0)) && (domain.c
      Real_t ratio ;
      Real_t olddt = domain.deltatime() ;

      /* This will require a reduction in parallel */
      Real_t gnewdt = Real_t(1.0e+20) ;
```

```
doLoop
  0x2c0: mov %edi,0xfffffffffffffff0(%rsp)
  0x2c4: movd 0xfffffffffffffff0(%rsp),%xmm7
  0x2ca: mov $0x1,%esi
  dosum
    mult1
      714: mov $0x1,%esi
      719: pcmpeqd %xmm0,%xmm0
    mult3
      719: pcmpeqd %xmm0,%xmm0
      724: movdqa 0x143(%rip),%xmm3
      733: mov 0x200d14(%rip),%rax
      740: movdqa 0x143(%rip),%xmm2
      749: pshufd %xmm7,$0x0,%xmm7
      754: movdqa 0x115(%rip),%xmm4
      763: movdqa %xmm7,%xmm1
      768: lea 0x1000(%rax),%rcx
      775: lea 0x401000(%rax),%r8
      782: psrlq $0x20,%xmm1 1, 0
      788: mov %esi,0xfffffffffffffff4(%rsp)
      792: movd 0xfffffffffffffff4(%rsp),%xmm4
      798: lea 0xfffffffffffffffe(%rsi),%edx
      801: movdqa %xmm4,%xmm2
    mult1
      806: mov %edx,0xfffffffffffffff0(%rsp)
      810: lea 0xfffff000(%rcx),%rdx
      817: pshufd %xmm4,$0x0,%xmm3
      822: movd 0xfffffffffffffff0(%rsp),%xmm4
      828: movdqa %xmm3,%xmm6
      832: pshufd %xmm4,$0x0,%xmm4
      837: psrlq $0x20,%xmm6
      842: movdqa %xmm4,%xmm5
```

Fig. 9. This prototype combines source code, an inlining tree showing inlined instructions, and a CFG View. The inlining tree shows all instructions associated with inlining. The CFG View shows all instructions in the nodes. The instructions obfuscate the structure in each view, so we removed them to focus the inlining tree and CFG on structure and navigation with a separate flat view of the diassembly.



```
/*
  This is a Version 2.0 MPI + OpenMP implementation c

                Copyright (c) 2010-2013.
      Lawrence Livermore National Security, LLC.
Produced at the Lawrence Livermore National Laborato
                  LLNL-CODE-461231
                 All rights reserved.

This file is part of LULESH, Version 2.0.
Please also read this link -- http://www.opensource.c

//////////////
DIFFERENCES BETWEEN THIS VERSION (2.x) AND EARLIER VE
* Addition of regions to make work more representativ
* Default size of each domain is 30^3 (27000 elem) in
  more representative of our actual working set sizes
* Single source distribution supports pure serial, pu
  and MPI+OpenMP
* Addition of ability to visualize the mesh using Vis
  https://wci.llnl.gov/codes/visit/download.html
* Various command line options (see ./lulesh2.0 -h)
 -q              : quiet mode - suppress stdout
 -i <iterations> : number of cycles to run
 -s <size>       : length of cube mesh along side
 -r <numregions> : Number of distinct regions (def: 1
 -b <balance>    : Load balance between regions of a
```

```
root
  CalcHourglassControlForElems
    0: push R15, RSP
    2: push R14, RSP
    4: push R13, RSP
    6: push R12, RSP
    8: push RBP, RSP
    9: push RBX, RSP
    10: sub RSP, 678
    17: mov EAX, [RDI + 578]
    23: mov [RSP + 290], RDI
    31: mov [RSP + 218], RSI
    39: movsd [RSP + 2a0], XMM0
    48: lea EBX, 0 + RAX * 8
    55: mov R14D, EAX
    58: mov [RSP + 260], EAX
    65: movsxd RBX, EBX
    68: shl/sal RBX, 3
    72: mov RDI, RBX
    75: call 0 + RIP + 5
    80: mov RDI, RBX
    83: mov [RSP + 220], RAX
    91: call 0 + RIP + 5
    96: mov RDI, RBX
    99: mov [RSP + 228], RAX
    107: call 0 + RIP + 5
    112: mov RDI, RBX
    115: mov [RSP + 230], RAX
    123: call 0 + RIP + 5
    128: mov RDI, RBX
    131: mov [RSP + 238], RAX
```

Fig. 10. This prototype iteration uses only the basic block IDs in the CFG nodes compared to the full instructions in Fig. 9. This change emphasized the topology and structure of the CFG, where multiple loops are now visible. Loop shading cues have not yet been added. Ultimately, we decided basic block ID was too abstract. The final version includes containing-function name.