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 corresponding code, and continued their analysis without pause.

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.
with nested four loops, and b) a “RAJA-sequential” (“RAJA”) version well.”

vector instructions (vfmadd231pd) for loop preamble and postamble instructions in the CFG View (Fig. 5, full context: (Fig. 4, ). P1 asked to scroll through the inlining tree. They remarked the code structure is similar to the base version, but obfuscated by the long call stack. They further identified candidates for loop unrolling.

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.

(P1) 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.

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 hypothesized that the top-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 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 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)).
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 loops. 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 pdatum, 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.
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.

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

```c
#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 doum(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] = doum(x, y+1);
        }
    }
}
```

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

```c
#include <limits.h>
#include <vector>
#include <stdio.h>
#include <string.h>
#include <typeinfo>
#include <time.h>
#include <sys/time.h>
#include <iostream>
#include <unistd.h>

#include "ilibrium.h"

#include "lulesh.h"

/* Work Routines */

static inline
void TimeIncrement(const void *domain) {
    Real_t targett = domain->stop.time - domain->start.time;
    if ((domain->end time() <= Real_t(0.0)) || (domain->target_ratio < 0.1)) {
        Real_t oldt = domain->delta.time();
        /* This will require a reduction in parallel */
        Real_t newdt = Real_t(1.0*10);
    }
```

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 disassembly.
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.