Improving Attribution of Performance Measurements for Optimized Code

John Mellor-Crummey and Mark Krentel

Department of Computer Science
Rice University

http://hpctoolkit.org

Petatools 2014 August 4, 2014

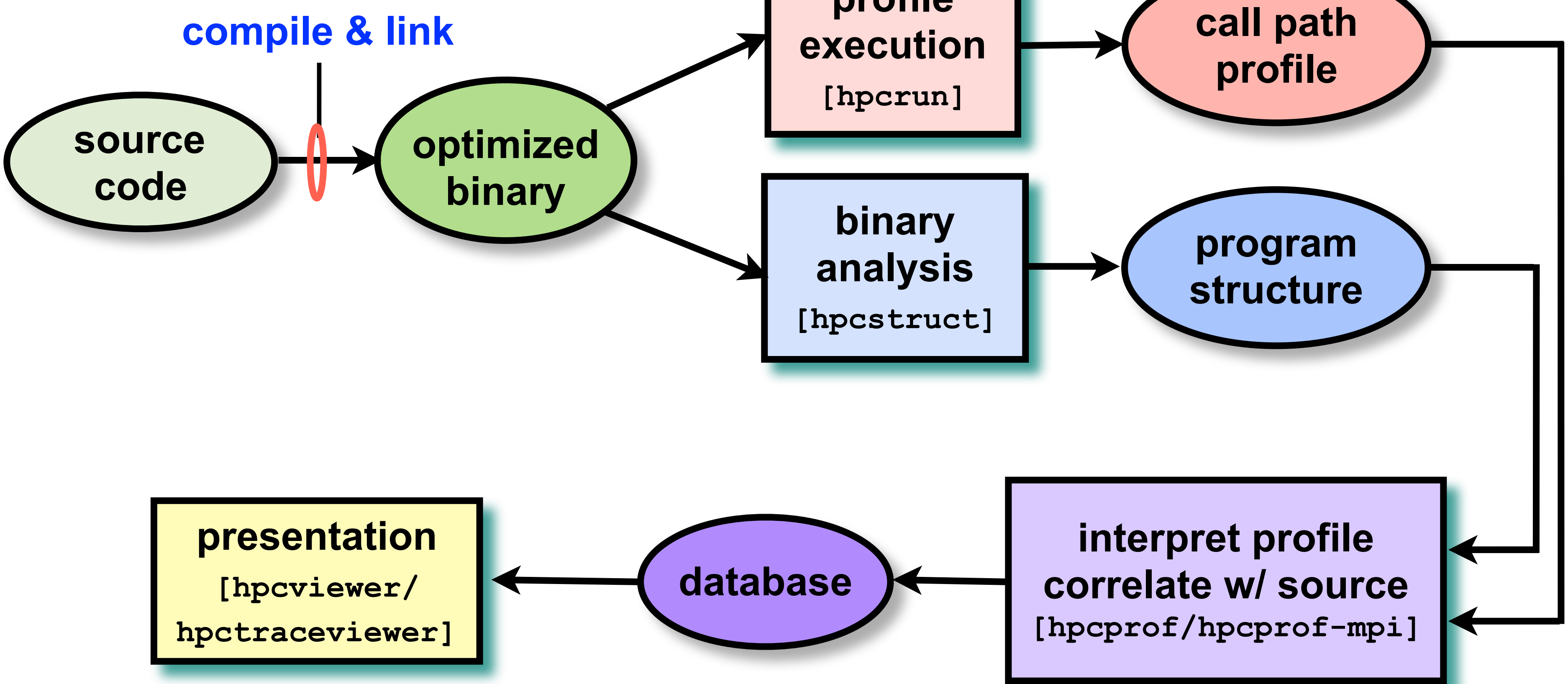
Motivation

- Modern software uses abstractions to manage complexity
  - procedures
  - classes
  - parameterized templates for algorithms and data structures

- Programmers rely on optimizing compilers to transform abstractions for efficient execution
  - compose algorithm and data structure templates
    - e.g., C++ Standard Template Library (STL), Boost, ...
  - inline procedures
  - transform loop nests

- Understanding the performance of modern software requires measuring the performance of optimized code and relating measurements back to the program source code

HPCToolkit Workflow

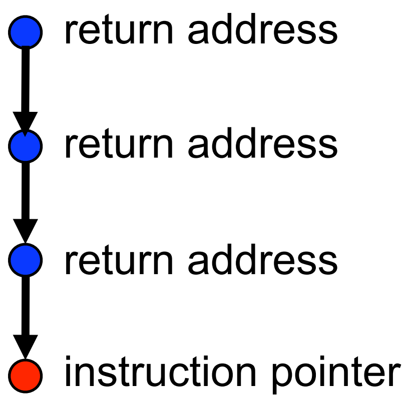


Call Path Profiling

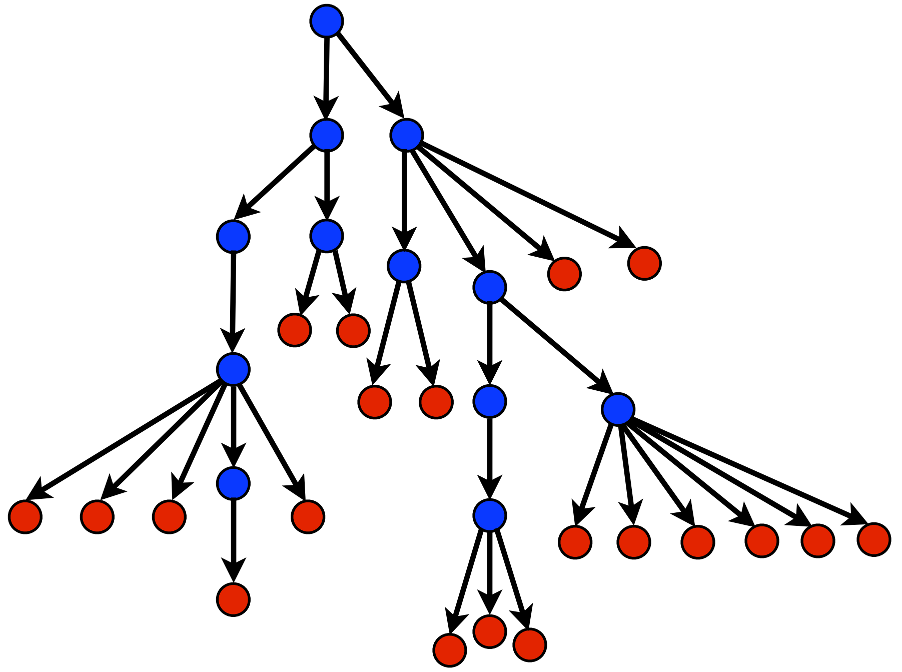
Measure and attribute costs in context

- sample timer or hardware counter overflows
- gather calling context using stack unwinding

Call path sample



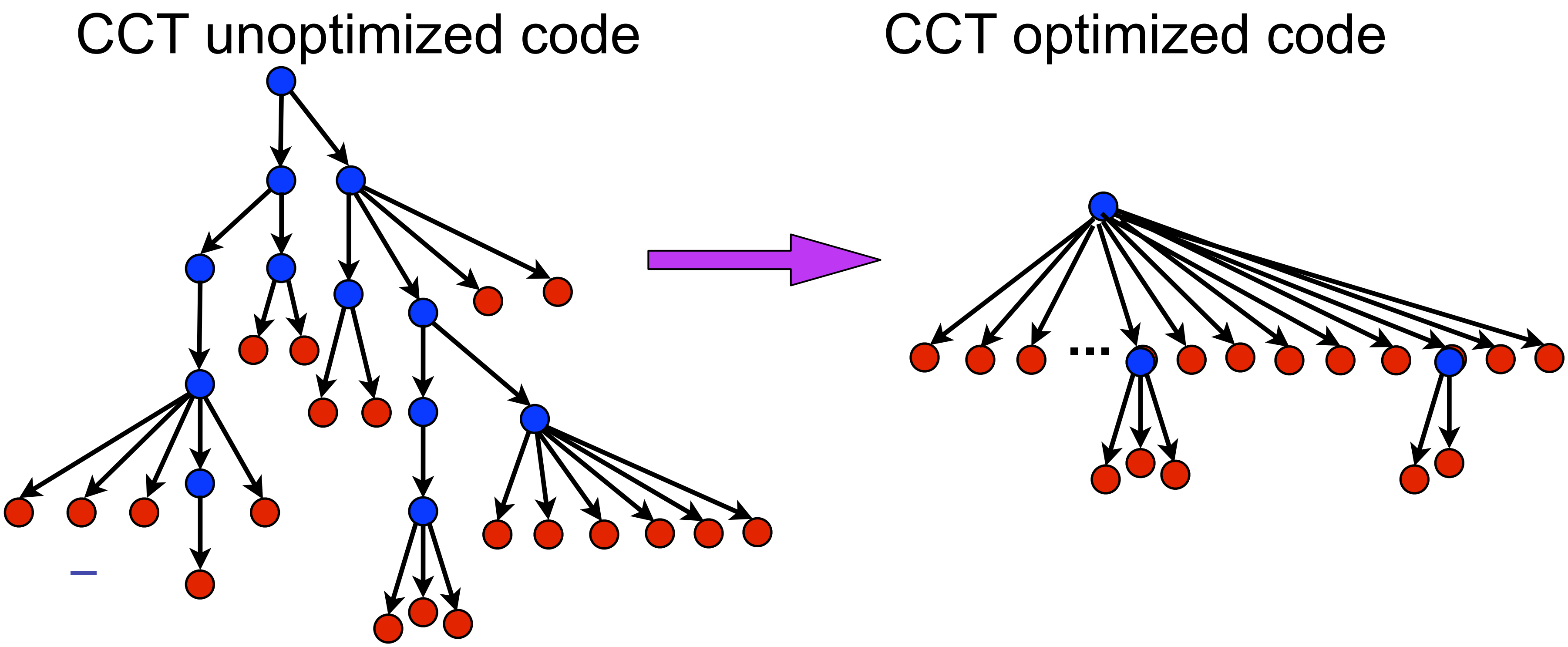
Calling context tree



Overhead proportional to sampling frequency...
...not call frequency

Understanding Optimized Code can be Difficult

- Structure of code is radically different after template instantiation, function inlining, and loop transformations
  - functions contain code from multiple files and functions



- Control flow graph structure is often rather complex
  - more than simple loops

Starting Point for This Work

Nathan Tallent, John Mellor-Crummey, and Michael Fagan. Binary analysis for measurement and attribution of program performance. PLDI '09. ACM, New York, NY, 441-452

- Binary analysis for call stack unwinding of unmodified optimized code
  - need to determine return address
  - parent's value for frame pointer register
- Binary analysis for attribution of performance to optimized code
  - identified inlined code as code from different source file
  - reported only one level of inlining
    - enclosing context
    - a single source line mapping for each generated instruction

An Example: small.cpp

```
using namespace std;
vector <int> v;
inline static void addToVector(int i) {
    v.push_back(i);
}
void do_work(int num) {
    v.clear();
    for (int i = 0; i < num; i++) {
     addToVector(i);
   }
}
int main(int argc, char **argv) {
    int len = 1000;
    int num, k;
    if (argc < 2 || sscanf(argv[1], "%d", &num) < 1) {
     num = 20;
    }
    num *= len;
    for (k = 0; k < num; k++) {
     do_work(len);
    }
    return 0;
}
```

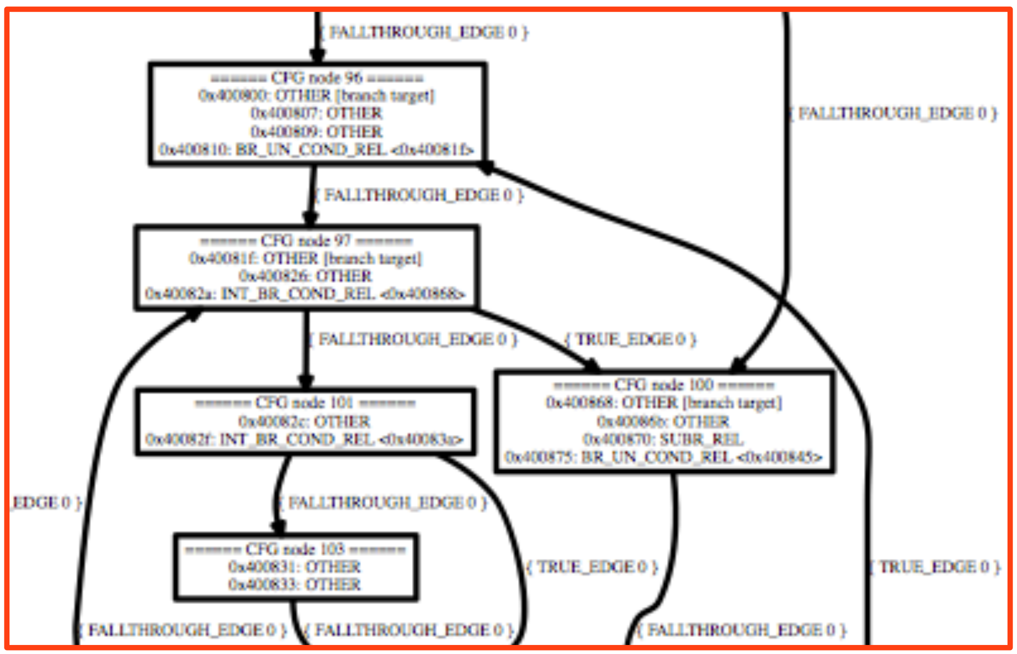
Generated Code for small.cpp (g++ 4.4.6)

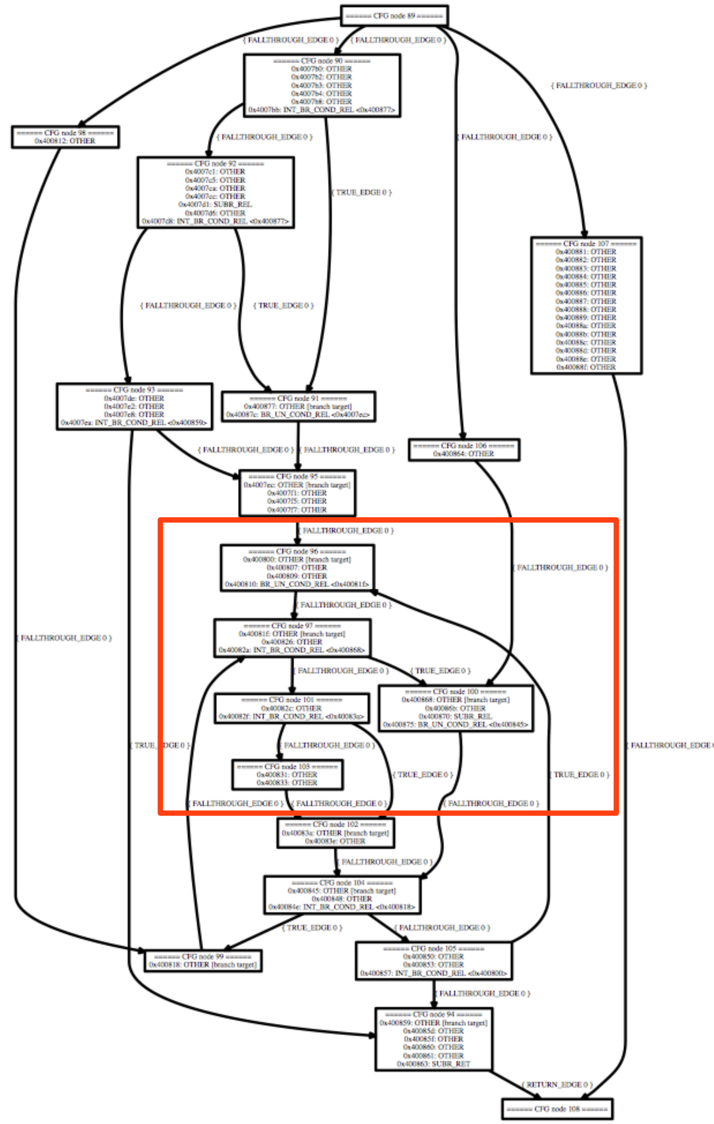
91 lines of assembly code for main

- Multiple levels of inlining
- Inlines the following functions
  - dowork
  - addToVector
  - vector::push_back
  - __gnu_cxx::new_allocator
  - vector::clear
  - vector::_M_erase_at_end
- Only two function calls left
  - iterator in push_back
  - sscanf

Construct the CFG

- Parse the machine code in an executable
- Build a CFG at the level of basic blocks





g++ 4.4.6

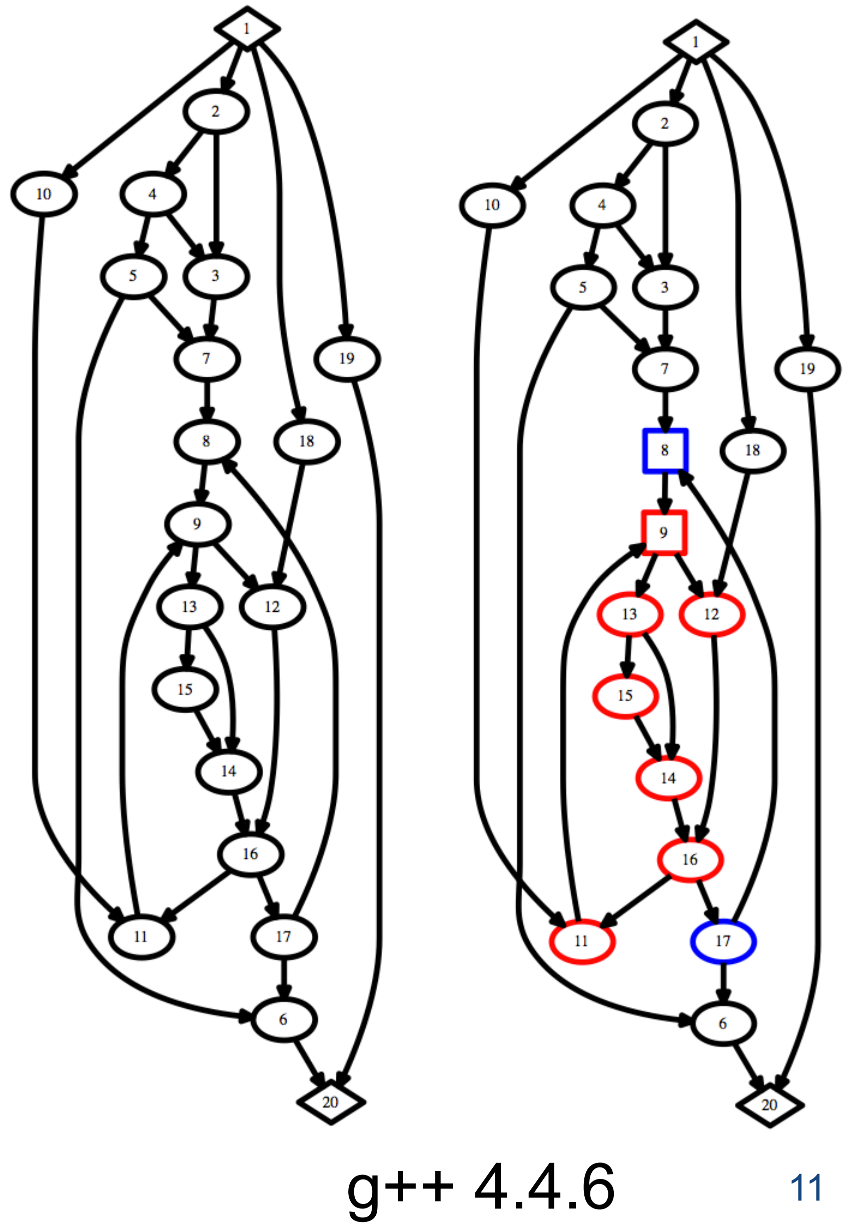
Identify Loops

Directed Graph G = (V, E)

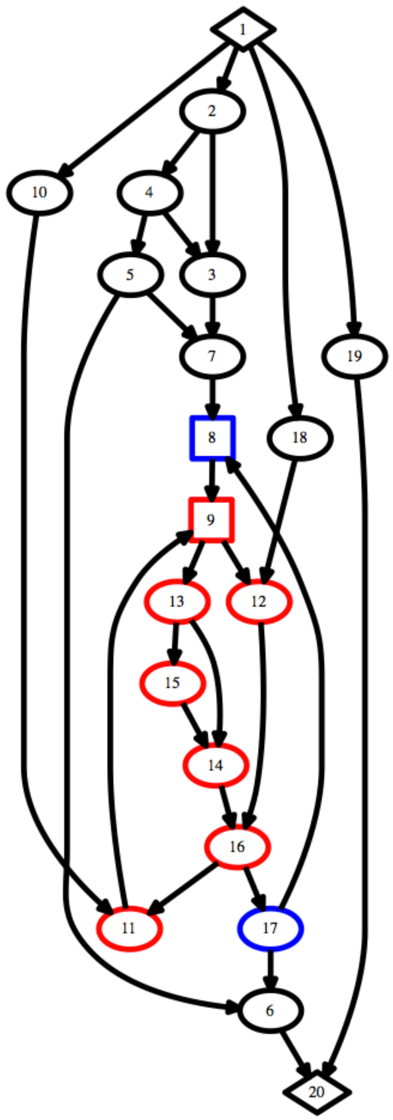
- Dominator
  - x dom y iff every execution path from entry to y goes through x
- Natural loop
  - defined by a back edge y ➔ x where x dom y
    - finds only single-entry loops
- Tarjan's algorithm finds single-entry, strongly-connected subgraphs
  - Robert Tarjan, "Depth-first search and linear graph algorithms," SIAM Journal on Computing 1(2):146–160, June 1972.
  - sketch
    - based on depth-first search
    - an SCC body includes nodes that reach a lower node then itself
    - loop head: node where lowest reachable is itself
  - complexity: O(V + E)

Coping with Irreducible Loops

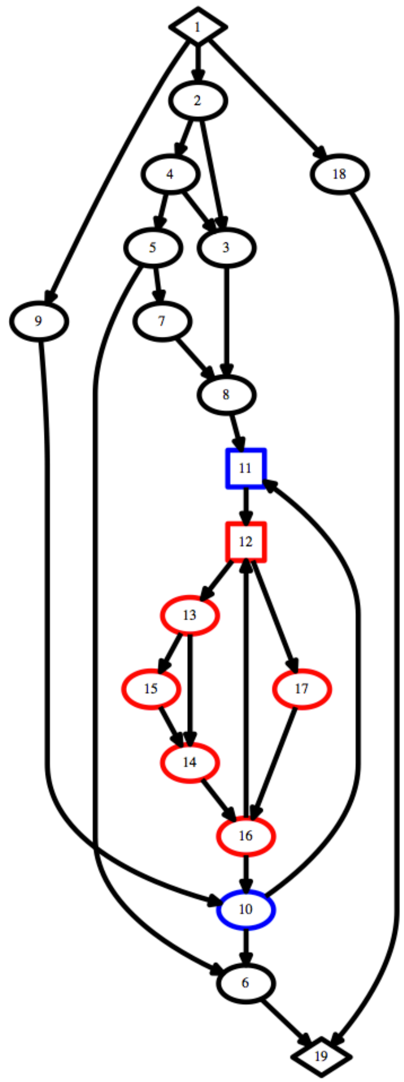
- Problem: not all cycles are single-entry loops
  - multiple entry loop: irreducible
- Paul Havlak. Nesting of reducible and irreducible loops. ACM TOPLAS 19(4):557-567, 1997.
  - uses definitions of reducible and irreducible loops which allows arbitrary nesting of either kind of loop
  - loop nesting tree can depend on the depth-first spanning tree used to build it
    - header node representing a reducible loop in one version of loop nesting tree can represent an irreducible loop in another



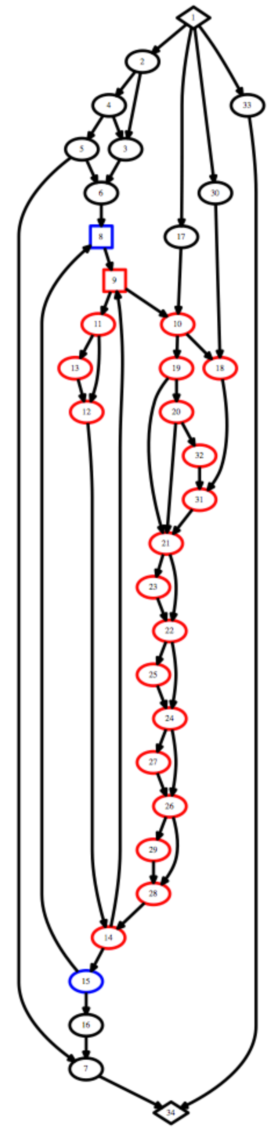
Considerable Variations in Code Shape



g++ 4.4.6



g++ 4.1.2



g++ 4.8.2

Challenges to CFG Construction

- Compiler optimizations make it difficult to recover accurate CFGs
  - tail calls
  - functions that don't return, e.g., exit, __cxa_throw, longjmp, ...
    - calls to through PLT to dynamically-linked routines
    - calls to routines statically-linked in a load module
- No indication of these features in DWARF
  - recover this info by processing /usr/include and C++ ABI headers

Tail Call Example from LLNL's LULESH

Fragment of source code

```
   if ( hgcoef > Real_t(0.) ) {
      CalcFBHourglassForceForElems(determ,x8n,y8n,z8n,dvdx,dvdy,dvdz,hgcoef);
   }

   Release(&z8n) ;
   Release(&y8n) ;
   Release(&x8n) ;
   Release(&dvdz) ;
   Release(&dvdy) ;
   Release(&dvdx) ;

   return ;
```

Sketch of generated code (gcc 4.4.6 -O3)

```
     if ( hgcoef > Real_t(0.) ) goto calc
rel: free(&z8n)
     free(&y8n)
     free(&x8n)
     free(&dvdz)
     free(&dvdy)
     push &dvdx
     jmp free
calc:inlined code for CalcFBHourglassForceForElems
     goto rel
```

Non-returning Function Example from miniFE

- Non-returning functions occur frequently, even in scientific codes
  - casting associated with inlined C++ I/O helper routines

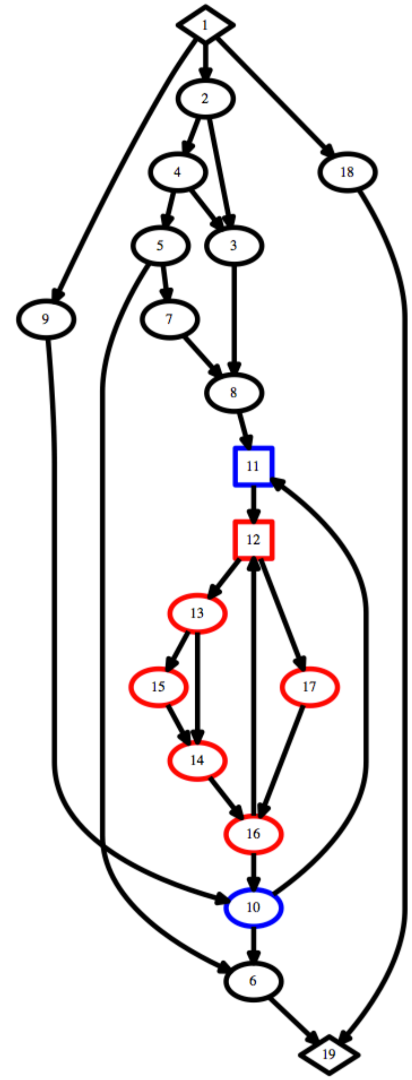
```
#ifndef _BASIC_IOS_H
...
_GLIBCXX_BEGIN_NAMESPACE(std)
  template<typename _Facet>
    inline const _Facet&
    __check_facet(const _Facet* __f)
    {
      if (!__f)
        __throw_bad_cast();
      return *__f;
    }
...
```

Mapping Back to Program Structure

- For each instruction, identify its full provenance
  - use DWARF info to recover complete static call chains
    - recover a full inlined call chain for each machine instruction
- Integrate information about loops and inlining to assemble a representation of static structure
- Not as simple as it sounds
  - where do loops belong in an inlined call chain?

Source Code Attribution for Loops

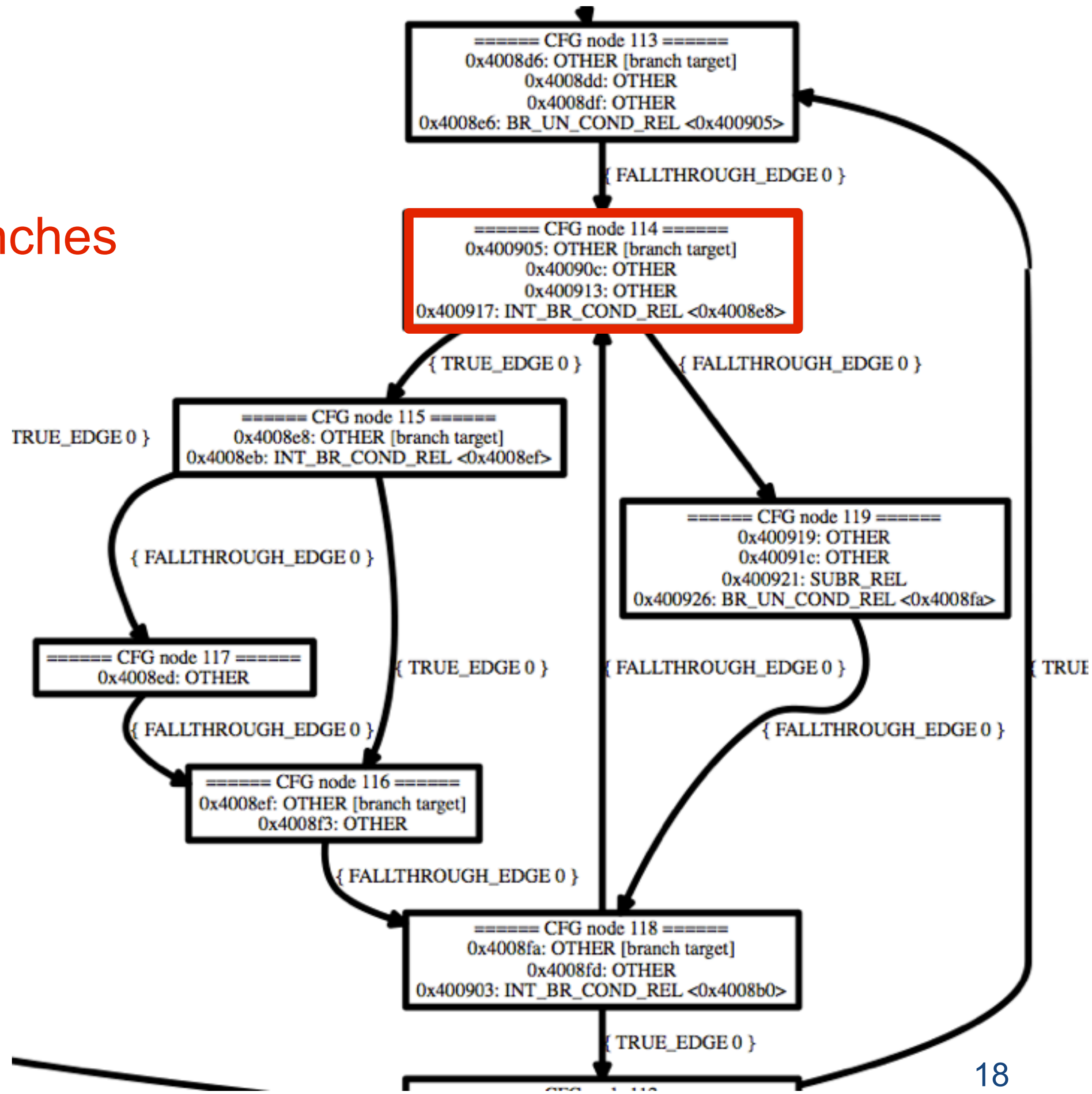
- Need to identify a source code position for each Interval and Irreducible interval
- What line number to use?
  - source line for first machine instruction in loop header?
  - source line for backward branch reaching loop header?
  - some complications ...
    - edges reaching loop header are not always backward branches



g++ 4.1.2

Detail of CFG for main (gcc 4.1.2)

Only fall through branches reach this header!



Associating a Loop with a Source Line

Today's heuristic

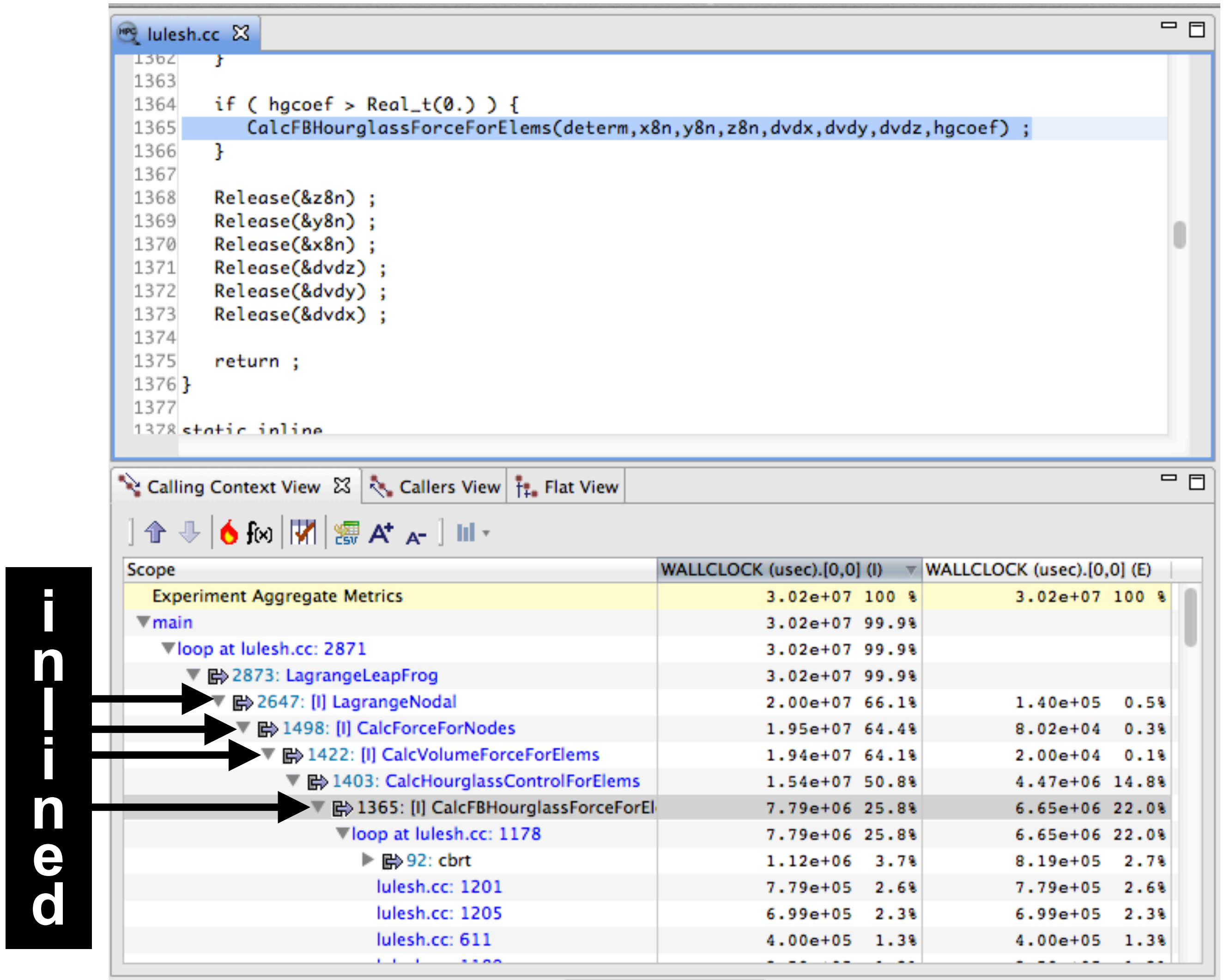
- Priority scheme
  - back edge
    - backward branch closing natural loop
  - true branches from within the loop
  - fall through edges from within the loop
- If none of these has a source mapping, use the mapping for the loop header
- If the source mapping for the loop header is less deeply nested than the source of the edge targeting it, use that instead

Assembling the Source View

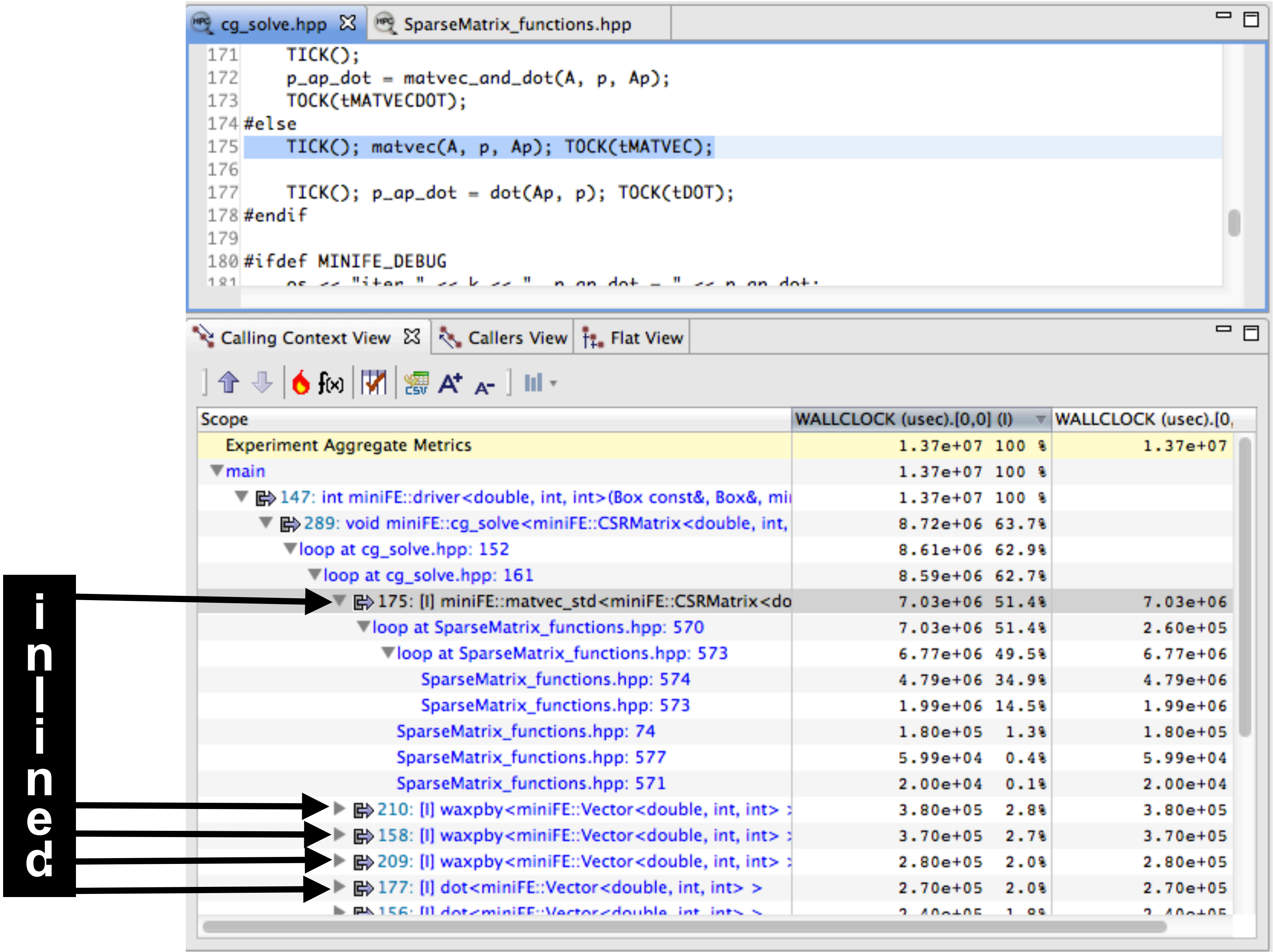
- Perform interval analysis of the CFG
- Recursively assemble the CCT for a procedure
  - for each interval
    - insert source code for all machine instructions inside into CCT
  - insert the call chain for the loop
    - never make the loop a child of any node inserted inside the loop
      - create copies of context where necessary
  - identify the least common ancestor between a loop and and the calling context for machine instruction inside it
    - treat copies of contexts along respective paths as equivalent
  - take the path below the LCA and insert that inside the loop
- For each “alien” context in inlined code, record information about
  - call site
  - callee
- Gracefully handle case where no static call chain information available
  - simply indicate that inlined code came from the following source file and line
- Present this in hpcviewer’s source code view as if real call chains, but indicate when function is inlined

LULESH: Attribution for Optimized Code

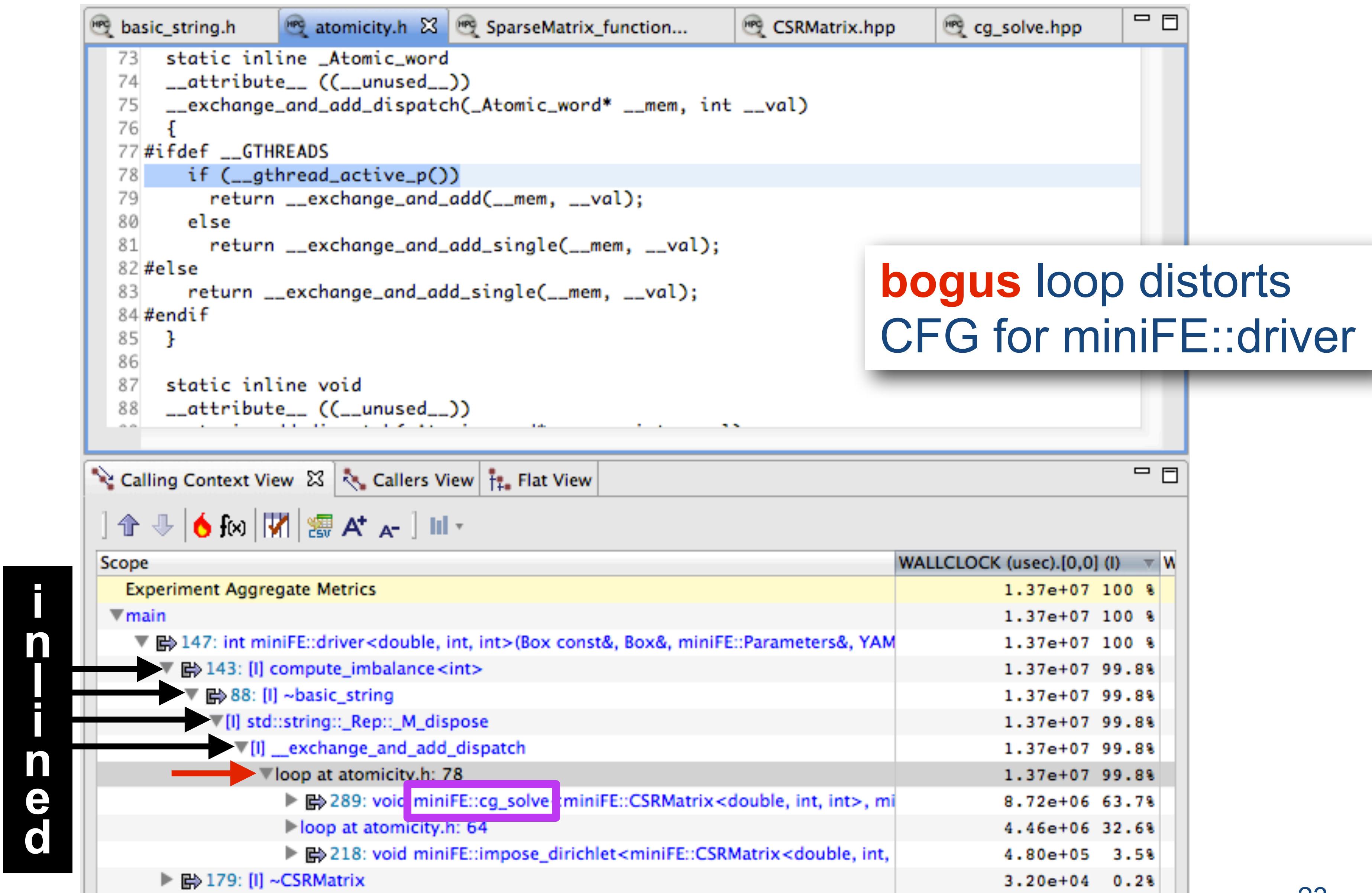
- Present full calling context and loops, as if an unoptimized executable



miniFE with Non-returning Function Analysis



miniFE without Non-returning Function Analysis



What's left?

- Technical issues
  - explore cases where embedding of loops in static call chains still isn't satisfactory
    - is there a better interpretation of the graph depending on depth first parse
    - can exhaustive analysis of a loop yield better results?
      - beyond just looking at loop header and incident edges
    - new 2007 flow graph analysis algorithm
      - better results?
      - better performance?
  - analysis speed for huge binaries?
- Community issues
  - lobby DWARF community to enhance standard with information about functions that don't return

Flowgraph Analysis References

- Robert Tarjan, “Depth-first search and linear graph algorithms,” SIAM Journal on Computing 1(2):146–160, June 1972.
- Paul Havlak. Nesting of reducible and irreducible loops. ACM TOPLAS 19(4): 557–567, July 1997.
- Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A New Algorithm for Identifying Loops in Decompilation. Static Analysis 14th International Symposium (SAS), LNCS 4634, pp. 170–183, 2007.