Lecture 17:
Performance Monitoring Tools

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2020

Scenario

- Student walks into office hours and says, “My code is slow / uses lots of memory / is SIGKILLED. I implemented X, Y, and Z. Are those good? What should I do next?”

- It depends.

What is my program doing?

- Measurements are more valuable than insights
  - Insights are best formed from measurements!

- We're Computer Scientists
  - We can write programs to analyze programs

Note about Examples

- The example programs in today's lecture are from Spring 2016 Assignment 3
  - OpenMP-based graph processing workload (paraGraph)
  - Millions to tens of millions of nodes
  - Code written for the GHC machines and Xeon Phi

My program is slow today.

- **What else is running?**
  - **Try "top"**

```
top - 14:43:26 up 25 days,  3:46, 50 users,  load average: 0.04, 0.05, 0.01
Tasks: 1326 total,   1 running, 1319 sleeping,   2 stopped,   4 zombie
Cpu(s):  0.0%us,  0.1%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16220076k total,  7646188k used,  8573888k free,   246280k buffers
Swap:  4194296k total,     3560k used,  4190736k free,  5219176k cached

   PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  2801 nobody     20   0  481m 3860 1192 S  1.0  0.0  63:45.33 gmetad
  3306 root       20   0  258m  11m 2128 S  0.7  0.1 161:54.86 lsi_mrdsnmpagen
  4920 nobody     20   0  297m  18m 3380 S  0.7  0.1 181:11.80 gmond
 49781 --------   20   0  106m 2144 1456 S  0.3  0.0   0:00.10 bash
 58119 bpr        20   0 15976 2220  936 R  0.3  0.0   0:00.30 top
106182 --------   20   0 24584 2184 1136 S  0.3  0.0   2:27.99 tmux
134225 --------   20   0  143m 1732  608 S  0.3  0.0   0:02.92 intelremotemond
...
```

What else can top tell us?

- CPU / Memory usage of our program
  - `./paraGraph kbfs com-orkut_117m.graph -t 8 -r`

```
top - 15:54:27 up 3 days, 23:58,  6 users,  load average: 3.43, 1.15, 0.43
Tasks: 286 total,   2 running, 284 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.8 us,  0.2 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  32844548 total, 31305468 used,  1539080 free,   435012 buffers
KiB Swap:  7999484 total,    13176 used,  7986308 free. 27364456 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
23457 bpr       20   0 1559584 979704   3420 R 796.4  3.0   0:27.91 paraGraph
 1071 root      20   0   75892   6560   5564 S   2.0  0.0  19:58.05 cups-brows+
21506 root      20   0   87680  17300   5460 S   0.7  0.1   1:08.43 cupsd
23408 bpr       20   0   24956   3196   2588 R   0.3  0.0   0:00.18 top
    1 root      20   0   36100   4204   2632 S   0.0  0.0   0:01.02 init
```

Do I have to use top?

- **No. Time was part of the assignment 3 qsub jobs.**

```
$ tail -n 1 bpr_grade_performance.job
time ./grade_performance.py ./$exe
```

- **time is often a shell command, there is also the time binary**

- ```
  /usr/bin/time ./paraGraph kbfs com-orkut_117m.graph -t 8 -r
  ...
  33.16user 0.10system 0:05.54elapsed 600%CPU
  (0avgtext+0avgdata 979708maxresident)k 0inputs+0outputs
  (0major+5624minor)pagefaults 0swaps
  ```
  ```

But why is it slow?

- Where is the time spent?
  - Put timing statements around probable issues
  - Print results

- OR
  - Use a tool to insert timing statements

Program Instrumentation

- **When to inject the instrumentation?**
  - **When the program is compiled.**
  - **When the program is run.**

Instrumentation Tool Families

- **Program Optimization**
  - **Gprof**
  - **Perf**
  - **VTune**
- Program Debugging
  - Valgrind
  - Sanitizers
- Advanced Analysis
  - Pin
  - Contech

Amdahl's Law Revisited

- 1- s – a component of the program
- p – speedup of that component

$$\text{speedup} \le \frac{1}{s + \frac{1-s}{p}}$$

- The more time something takes
  - The more speedup small improvements make
- Concentrate program optimization on:
  - Hot code
  - Common cases

GProf

- Enabled with “-pg” compiler flag
- Places a call into every function
  - Calls record the call graph
  - Calls record time elapsed

- Run the program.
- Run gprof <prog name>

GProf cont

- **Output shows both the total time in each function**
  - **And cumulative time in calling trees**

- **Can be useful with large call graphs**

- `$./paraGraph pagerank -t 8 -r soc-pokec_30m.graph`
- `$gprof`

```
%    cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
69.35      0.43     0.43        1   430.00   430.00  build_incoming_edges(graph*)
30.65      0.62     0.19       18    10.56    10.56  pagerank(graph*, ...)
 0.00      0.62     0.00  1632803     0.00     0.00  addVertex(VertexSet*, int)
 0.00      0.62     0.00        7     0.00     0.00  newVertexSet(T, int, int)
 0.00      0.62     0.00        7     0.00     0.00  freeVertexSet(VertexSet*)
```

Perf

- Modern architectures expose performance counters
  - Cache misses, branch mispredicts, IPC, etc

- Perf tool provides easy access to these counters
  - perf list – list counters available on the system
  - perf stat – count the total events
  - perf record – profile using one event
  - perf report – Browse results of perf record

- Perf is present on GHC machines tested

Perf stat

- Can be run with specific events or a general suite

- perf stat [-e …] app
  - Many counters come in pairs, each needs a separate -e
    - cycles, instructions
    - branches, branch-misses
    - cache-references, cache-misses
    - stalled-cycles-frontend
    - stalled-cycles-backend
  - Processors can only enable ~4 counters, else it must multiplex

Perf stat (default) output

```
./paraGraph -t 8 -r pagerank /afs/cs/academic/class/15418-s16/public/asst3_graphs/soc-
pokec_30m.graph':

  2366.633970      task-clock (msec)         #    1.758 CPUs utilized
          109      context-switches          #    0.046 K/sec
            9      cpu-migrations            #    0.004 K/sec
        6,168      page-faults               #    0.003 M/sec
7,513,900,068      cycles                    #    3.175 GHz                   (83.23%)
6,327,732,886      stalled-cycles-frontend   #   84.21% frontend cycles idle  (83.42%)
4,019,403,839      stalled-cycles-backend    #   53.49% backend  cycles idle  (66.86%)
3,222,030,372      instructions              #    0.43  insns per cycle
                                             #    1.96  stalled cycles per insn (83.43%)
  457,170,532      branches                  #  193.173 M/sec                 (83.30%)
   12,354,902      branch-misses             #    2.70% of all branches       (83.24%)
```

So what is the bottleneck?

More perf stat

- **Maybe memory is a bottleneck.**

```
201,493,787    cache-references
 49,347,882    cache-misses      #   24.491 % of all cache refs
```

- **24% misses, that's not good.**
- **But what should we do?**

Perf record

- Pick an event (or use the default cycles)

- When the event's counter overflows
  - The processor sends an interrupt
  - The kernel records where (PC value) of the program

- NOTE: counters update in funny, microarchitectural ways so intuition may be required

"Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it."

Perf cache misses

- **Are cache misses the problem?**
  - **Sort of.**

```
Samples: 11K of event 'cache-misses', Event count (approx.):
181771931

Overhead  Command    Shared Object        Symbol
  47.18%  paraGraph  paraGraph        [.] edgeMapS<State<float> >
  46.84%  paraGraph  paraGraph        [.] build_incoming_edges
   2.70%  paraGraph  [unknown]        [k] 0xffffffff813b2537
   1.37%  paraGraph  [unknown]        [k] 0xffffffff813b2915
```

Perf report cycles

- perf report shows analysis from record
  - Commandline interactive interface

```
Samples: 13K of event 'cycles', Event count (approx.): 11108635969
Overhead  Command    Shared Object                Symbol
  65.93%  paraGraph  paraGraph              [.] edgeMapS<State<float> >
  27.66%  paraGraph  paraGraph              [.] build_incoming_edges
   1.85%  paraGraph  paraGraph              [.] vertexMap<Local<float> >
   1.02%  paraGraph  [kernel.kallsyms]      [k] clear_page_c
   0.88%  paraGraph  paraGraph              [.] addVertex
   0.60%  paraGraph  [kernel.kallsyms]      [k] copy_user_generic_string
```

- Over 25% of program time is in creating the graph
  - This also skews the perf stats

Deep dive

- **Selecting a function will display its assembly with function-local %**

```
      |        bool update(Vertex s, Vertex d)
      |        {
      |          float add = pcurr[s] / outgoing_size(graph, s);
 2.97 |        divss  %xmm1,%xmm0
 5.22 |        jmp    162
      |        nop
      |160:    mov    %eax,%edx
      |          #pragma omp atomic
      |          pnext[d] += add;
 0.16 |162:    mov    %edx,0x18(%rsp)
 1.28 |        mov    %edx,%eax
 0.01 |        movss  0x18(%rsp),%xmm2
 2.71 |        addss  %xmm0,%xmm2
 4.63 |        movss  %xmm2,0x18(%rsp)
 1.16 |        mov    0x18(%rsp),%r15d
 3.99 |        lock   cmpxchg %r15d,(%rcx)
25.22 |        cmp    %eax,%edx
      |        jne    160
```

1. OMP atomic -> lock cmpxchg
2. This instruction is 25%*65% of execution time

Deep dive 2

- **kBFS is really, really slow. Why?**

```
Samples: 48K of event 'cycles', Event count (approx.):
39218498652

Overhead  Command    Shared Object                    Symbol
  63.78%  paraGraph  paraGraph        [.] edgeMapS<RadiiUpdate>
  19.33%  paraGraph  paraGraph        [.] edgeMap<RadiiUpdate>
   8.21%  paraGraph  paraGraph        [.] build_incoming_edges
   3.88%  paraGraph  paraGraph        [.] vertexMap<VisitedCopy>
```

- **That's almost all my code. :(**

Disassemble it!

- **What is taking all of kbfs's time?**

```
bool update(Vertex src, Vertex dst) {
       |           bool changed = false;
       |           for (int j = 0; j < NUMWORDS; j++) {
       |             if (visited[dst][j] != visited[src][j]) {
  0.11 |       mov    0x0(%r13),%rax
  0.21 |       mov    (%rax,%rdi,1),%rbp
  0.20 |       mov    (%rax,%rcx,8),%rax
 14.88 |       mov    0x0(%rbp),%ebp
  1.15 |       mov    (%rax),%eax
 68.27 |       cmp    %eax,%ebp
  0.02 |       je     108
       |        // word-wide or
       |        __sync_fetch_and_or(&(nextVisited[dst][j]), visited[dst]
  1.54 |       mov    0x8(%r13),%rcx
  0.34 |       or     %eax,%ebp
  0.02 |       mov    (%rcx,%rdi,1),%rcx
  0.31 |       lock   or     %ebp,(%rcx)
       |               int oldRadius = radii[dst];
       |               if (radii[dst] != iter) {
  6.45 |       mov    0x18(%r13),%ebp
```

VTune

- Part of Intel's Parallel Studio XE
  - Requires (free student) license from Intel

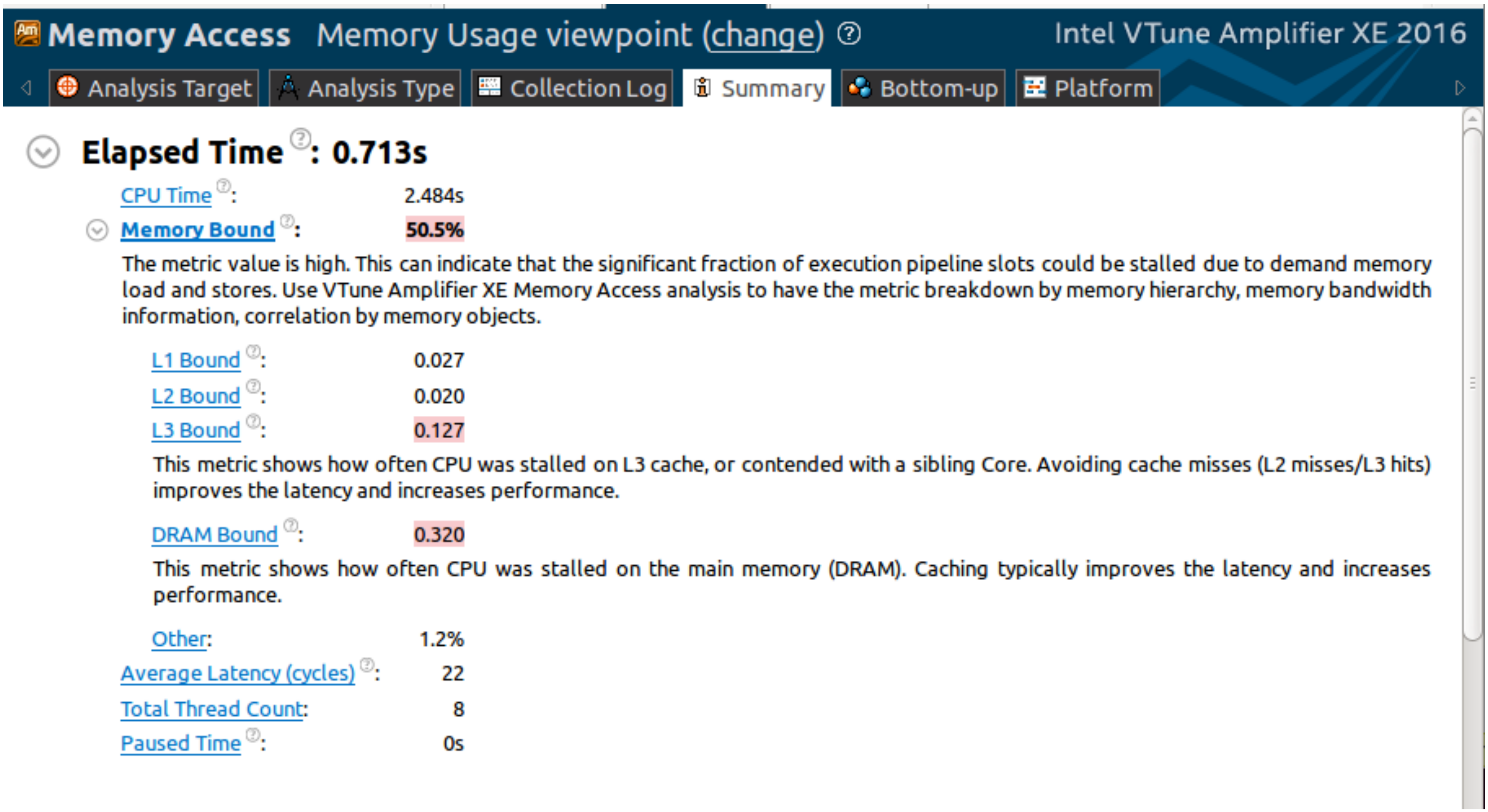
- Similar to perf
  - Also includes analysis across related counters

VTune Memory Bound

- That Spring, I asked many students in office hours:
  - "Do you think the graph code is memory bound?"

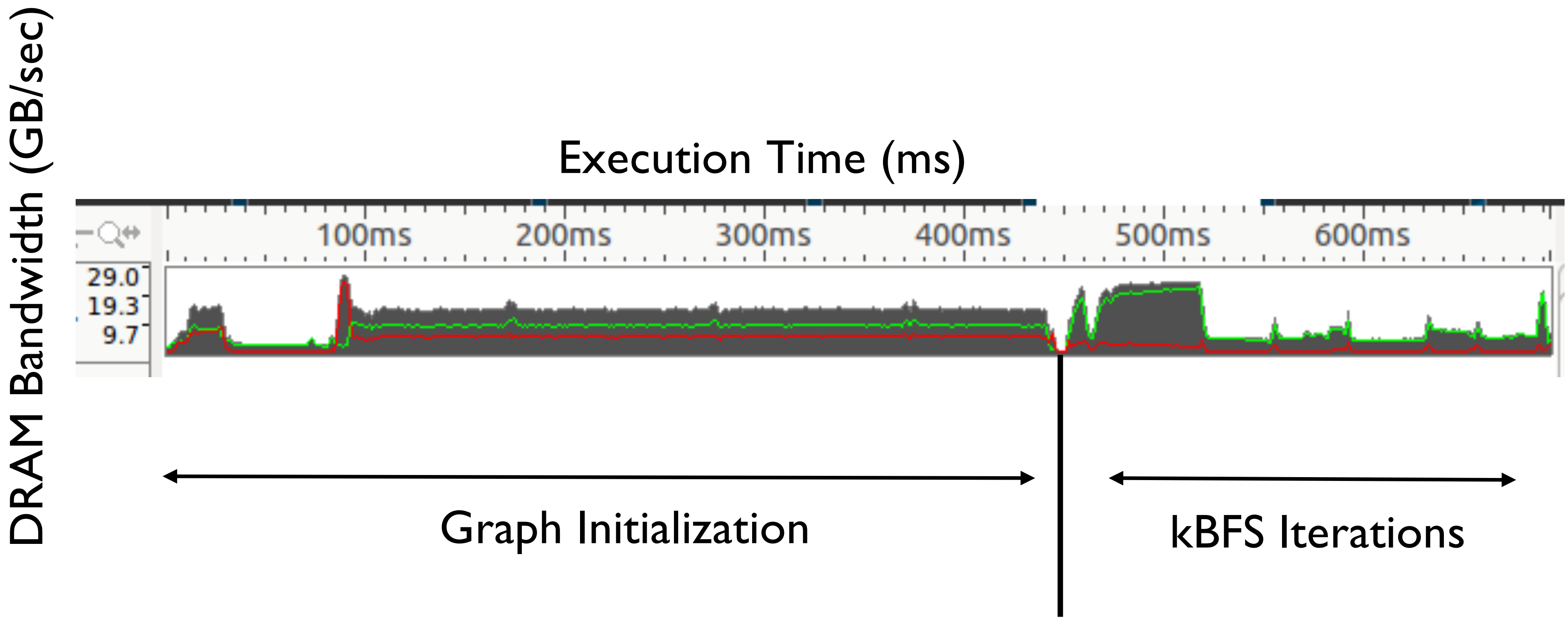
- Let's find out!
  - Create a project (select program + arguments to analyze)
  - Create an analysis
    - Microarchitecture -> Memory Access Analysis
  - Start!

Memory Access Analysis Results



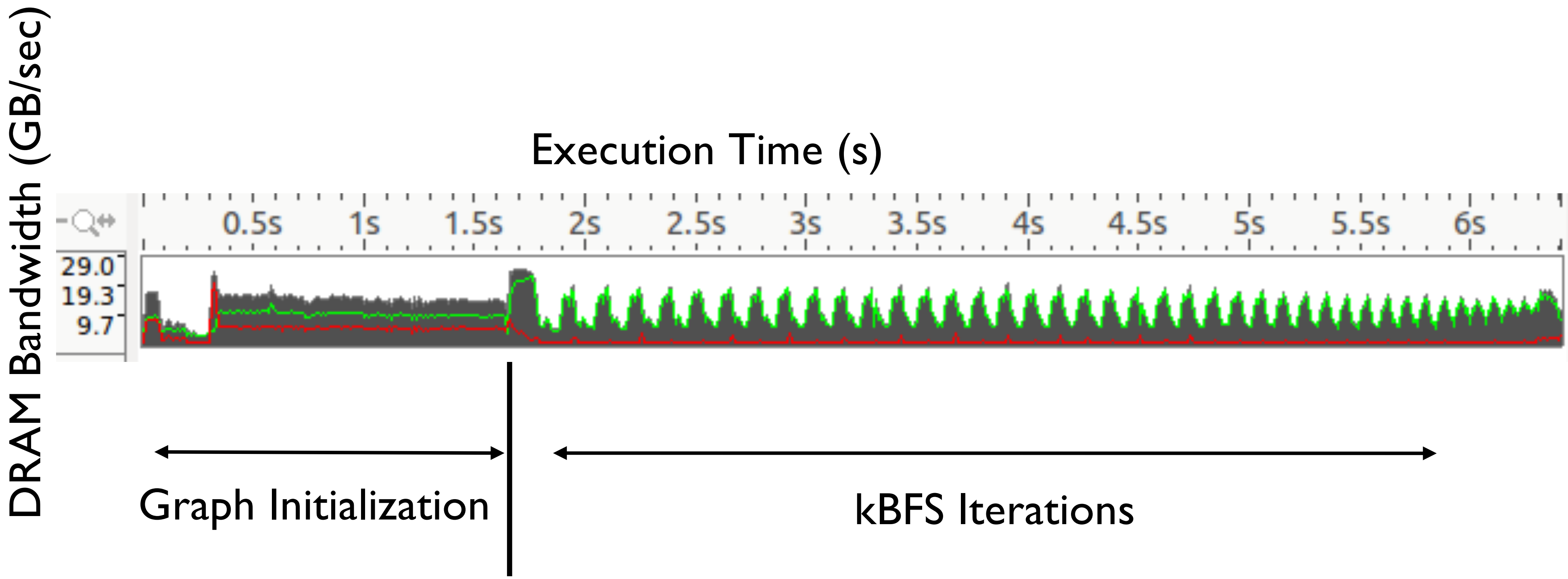
Further Analysis

- **Input: soc-pokec…**



Further Analysis

- **Input: com-orkut**



Instrumentation Tool Families

- Program Optimization
  - Gprof
  - Perf
  - VTune
- **Program Debugging**
  - **Valgrind**
  - **Sanitizers**
- Advanced Analysis
  - Pin
  - Contech

Valgrind

- **Heavy-weight binary instrumentation**
  - **Designed to shadow all program values: registers and memory**
  - **Shadowing requires serializing threads**
  - **4x overhead minimum**

- **Comes with several useful tools**
  - **Usually used for memcheck**

Valgrind memcheck

- Validates memory operations in a program
  - Each allocation is freed only once
  - Each access is to a currently allocated space
  - All reads are to locations already written
  - 10 – 20x overhead
- valgrind --tool=memcheck <prog …>

```
...
==29991== HEAP SUMMARY:
==29991==     in use at exit: 2,694,466,576 bytes in 2,596 blocks
==29991==   total heap usage: 16,106 allocs, 13,510 frees, 3,001,172,305 bytes allocated
==29991==
==29991== LEAK SUMMARY:
==29991==    definitely lost: 112 bytes in 1 blocks
==29991==    indirectly lost: 0 bytes in 0 blocks
==29991==      possibly lost: 7,340,200 bytes in 7 blocks
==29991==    still reachable: 2,687,126,264 bytes in 2,588 blocks
==29991==         suppressed: 0 bytes in 0 blocks
```

Address Sanitizer

- Compilation-based approach to detect memory issues
  - GCC and LLVM support
  - ~2x overhead

- Add "-fsanitize=address", make clean …

```
==1902== ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7f683e4c008c
at pc 0x41cb77 bp 0x7f683bc14a20 sp 0x7f683bc14a18
READ of size 4 at 0x7f683e4c008c thread T6
    #0 0x41cb76 (paraGraph+0x41cb76)
    #1 0x7f6852efdf62 (/usr0/local/lib/libiomp5.so+0x89f62)
    #2 0x7f6852ea7ae3 (/usr0/local/lib/libiomp5.so+0x33ae3)
    #3 0x7f6852ea620a (/usr0/local/lib/libiomp5.so+0x3220a)
    #4 0x7f6852ecab80 (/usr0/local/lib/libiomp5.so+0x56b80)
    #5 0x7f684fdb7b97 (/usr/lib/x86_64-linux-gnu/libasan.so.0.0.0+0x18b97)
    #6 0x7f684efa4181 (/lib/x86_64-linux-gnu/libpthread-2.19.so+0x8181)
    #7 0x7f684f2b447c (/lib/x86_64-linux-gnu/libc-2.19.so+0xfa47c)
...
```

Instrumentation Tool Families

- Program Optimization
  - Gprof
  - Perf
  - VTune
- Program Debugging
  - Valgrind
  - Sanitizers
- **Advanced Analysis**
  - **Pin**
  - **Contech**

Pin

- CompArch research project, now Intel tool
- Binary instrumentation tool framework
  - "Low" overhead
  - Provides many sample tools

- Given its architecture roots, it is best suited to specific architectural questions about a program
  - What is the instruction mix?
  - What memory addresses does it access?

Pin cont.

- Pin acts as a virtual machine
  - It reassembles the instructions with appropriate instrumentation

- Each "pintool" requests specific instrumentation
  - On basic block entry, record the static instruction count
  - On every memory operation, record the address
  - …

(Pin) Instrumentation Granularity

- Instruction
- Basic Block
  - A sequence of instructions
  - Single entry, single exit
  - Terminated with one control flow instruction
- Trace
  - A sequence of executed basic blocks
  - May span multiple functions

Pintool Example Instruction Count

- For every basic block in an identified trace
  - Insert somewhere in the block an instrumentation call to my routine
  - Pass my routine two arguments: number of instructions, thread ID

```
// Pin calls this function every time a new basic block is encountered.
// It inserts a call to docount.
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block  in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount for every bbl, passing the number of
instructions.

        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)docount,
IARG_FAST_ANALYSIS_CALL, IARG_UINT32, BBL_NumIns(bbl), IARG_THREAD_ID, IARG_END);
    }
}
```

Pintool Instruction Count Output

- $ pin -t pin/source/tools/ManualExamples/obj-intel64/inscount_tls.so -- ./paraGraph bfs -t 8 -r soc-pokec_30m.graph
- $ cat inscount_tls.out

```
Total number of threads = 9
Count[0]= 561617530
Count[1]= 16153
Count[2]= 44659367
Count[3]= 44863462
Count[4]= 44436576
Count[5]= 44458686
Count[6]= 43808683
Count[7]= 44055917
Count[8]= 43408645
```

Pin Cache Example

- … -t source/tools/Memory/obj-intel64/dcache.so …
- cat dcache.out

```
PIN:MEMLATENCIES 1.0. 0x0
#
# DCACHE stats
#
# L1 Data Cache:
# Load-Hits:         131764147   59.69%
# Load-Misses:        88995193   40.31%
# Load-Accesses:     220759340  100.00%
#
# Store-Hits:         71830273   71.07%
# Store-Misses:       29242668   28.93%
# Store-Accesses:    101072941  100.00%
#
# Total-Hits:        203594420   63.26%
# Total-Misses:      118237861   36.74%
# Total-Accesses:    321832281  100.00%
```

Pin Trace Example

- From a prior project
  - Records the instruction count
  - Records read/write and the address
- The trace was then used by a simulator

```
// Print a memory write record and the number of instructions between
// previous memory access and this access
VOID RecordMemWrite(UINT32 thread_id, VOID * addr)
{
  // format: W - [total num ins so far] - [num ins between prev mem access and this
access] - [address accessed]
  total_counts[thread_id]++;
  files[thread_id] << "W " << total_counts[thread_id] << " " <<  icounts[thread_id] <<
" " << addr << std::endl;
  reset_count(thread_id);
}
```

Contech

- Compiler-based instrumentation
  - Uses Clang and LLVM
  - Record control flow, memory accesses, concurrency

- Multi-language: C, C++, Fortran
- Multi-runtime: pthreads, OpenMP, Cilk, MPI
- Multi-architecture: x86, ARM

Contech continued

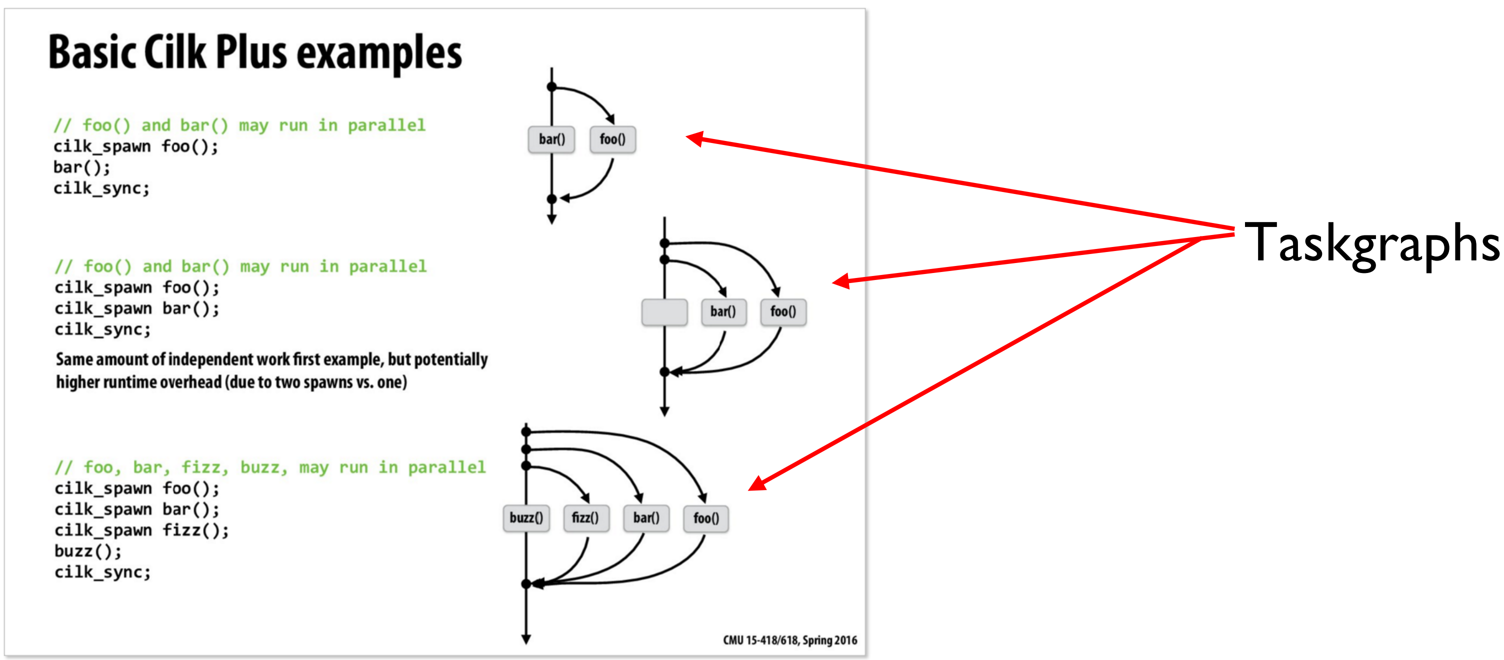
- Designed around writing analysis not instrumentation
  - All instrumentation is always used
  - Assumes the program is correct
  - Traces are analyzed after collection, not during

- Sample backends (i.e., analysis tools) are available
  - Cache Model
  - Data race detection
  - Memory usage

Contech Trace Collection

- Running the instrumented program generates a trace
  - Traces are processed into taskgraphs
  - Taskgraphs store the ordering of concurrent work

Perf Optimization I: Work Distribution and Scheduling



Contech Trace Collection Example

- `./paraGraph bfs -t 8 -r soc-pokec_30m.graph`
  - BFS Time: 0.0215s -> 0.2108s (9.8x slowdown)
  - 1855MB trace -> 1388MB taskgraph
    - 91 million basic blocks
    - 321 million memory accesses
    - 3 million synchronization operations

Summary Questions

- **If you may have a performance issue:**
  - **Is the issue reproducible?**
    - **Do you have a workload?**
    - **Is the system stable?**
  - **Is the workload at full CPU?**
    - **If not, are there other users / processes running?**
    - **Or does the workload rely heavily on IO?**
  - **Is the CPU time confined to a small number of functions?**
    - **What is the most time consuming function(s)?**
    - **What is their algorithmic cost and complexity?**

time / top

gprof / perf

Summary Continued

- You have a reproducible, stable workload
  - The machine is otherwise idle
  - The workload is fully using its CPUs
  - The algorithms are appropriate

- Is there a small quantity of hot functions?
  - Are their cycles confined to specific functions?
  - Are the costs of the instructions understood?

perf / VTune

Instrumentation Tool Links

- Gprof - https://sourceware.org/binutils/docs/gprof/
- Perf - https://perf.wiki.kernel.org/index.php/Main_Page
- VTune - https://software.intel.com/en-us/qualify-for-free-software/student
- Valgrind - http://valgrind.org/
- Sanitizers - https://github.com/google/sanitizers
- Pin - https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
- Contech - http://bprail.github.io/contech/

Other links

- **Performance Anti-patterns: http://queue.acm.org/detail.cfm?id=1117403**