Profiling Tutorial

Soon Yung Jun (Fermilab, SCD/Physics and Detector Simulation Group)

LArSoft Tools and Technology Workshop

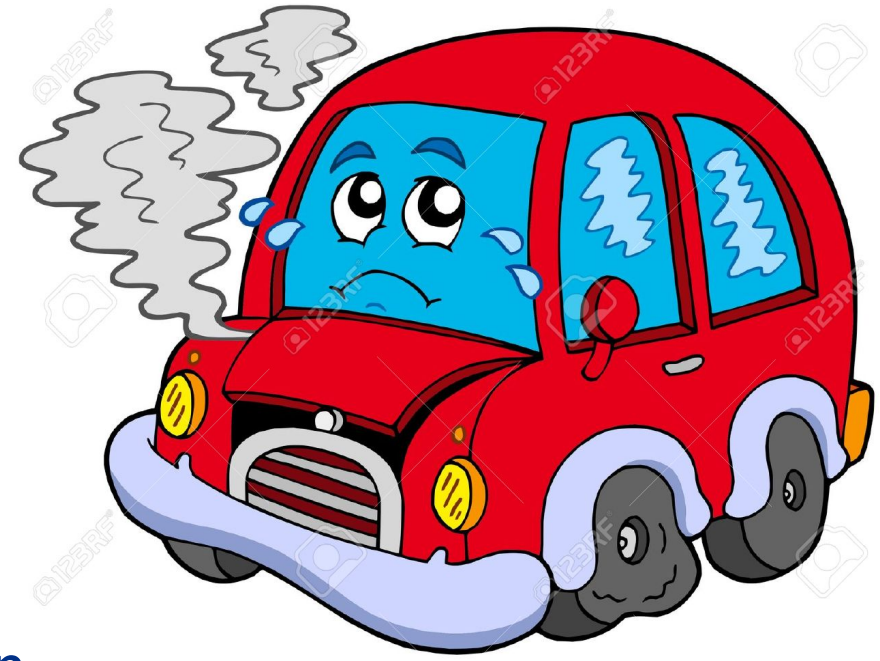
20 June 2017, Fermilab

Debugging is finally done! Ready for a test drive?

My program runs, but seems very slow …

CPU
Throughput
Instruction stall

Memory
Latency
Cache misses

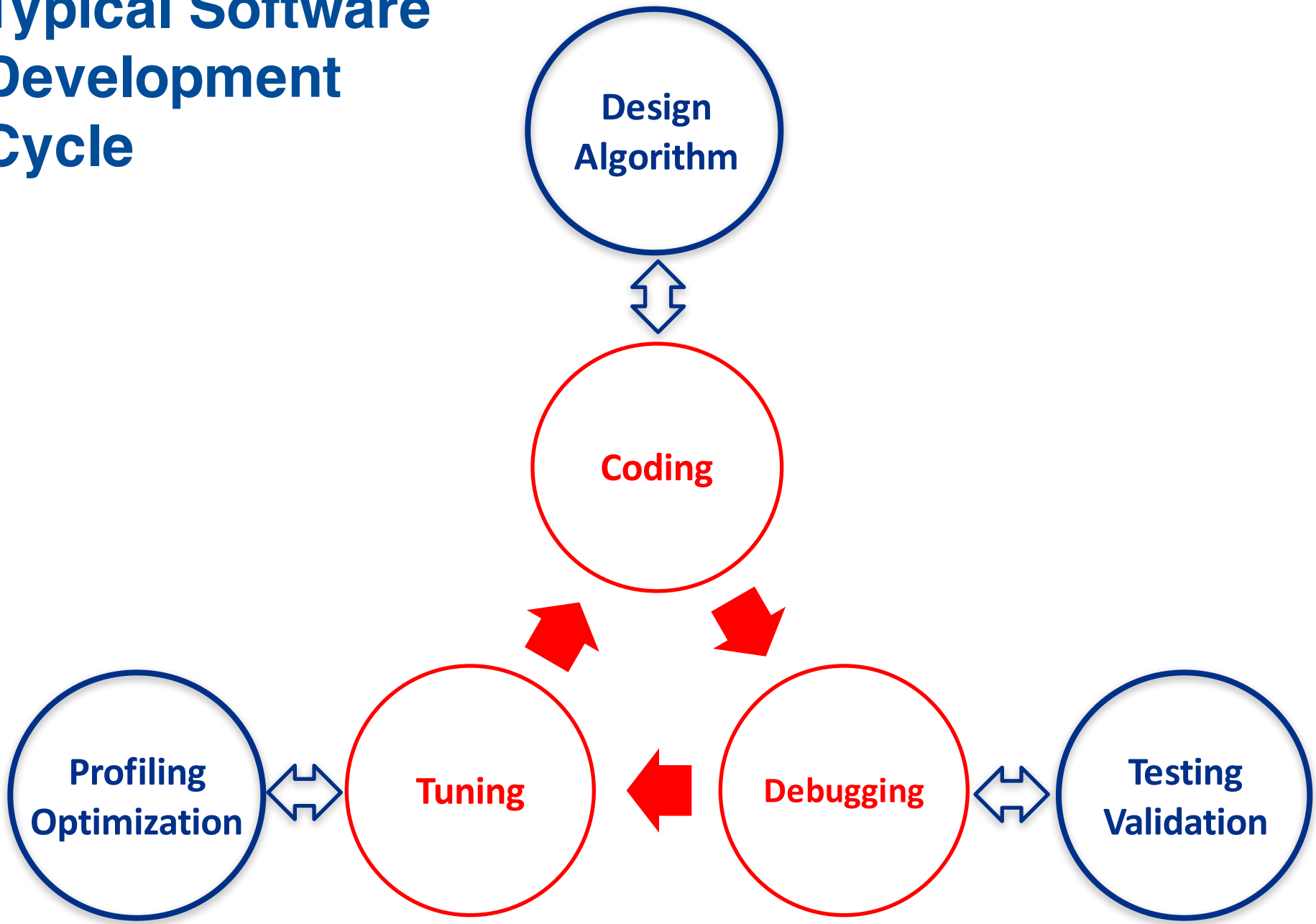


I/O
Communication
DB contention

Multithreading
Load balancing
Scalability

Where to start?

Typical Software Development Cycle



This Tutorial

- PART-I
  - A brief introduction to computing performance profiling
  - An overview of selected profiling tools and examples
- PART-II
  - Profiling results of the LArTest application with IgProf and OpenISpeedshop
- Demos and Questions

Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

PART-I

Introduction to Computing Performance Profiling

Overview of Selected Profiling Tools and Examples

Why Profile?

Performance tuning is an essential part of the development cycle

- Free lunch is over as modern hardware architectures are getting more complex and parallel
- HEP applications are usually complicated too
- Every $/Watt matters (computing with a limited budget)
- Understanding the code performance is responsibility of the software developer
- Maximize CPU flop rate and minimize memory operations (balancing them is not an easy task)

Computing Performance Profiling and Analysis

- Performance benchmarking quantifies usage/changes of CPU time and memory (amount required or churn)
- Performance profiling analyzes
  - Hot spots, bottlenecks and efficient utilization of resources
  - Code efficiency (instruction/cycle, latencies, I/O and etc.)
- Identifying opportunities for optimization

Low Hanging Fruit

After Production

Understanding Computer Performance

- Hardware platform (processors) popularly used in HEP
  - CISC (x86), RISC (ARM), MIC, GPU(SIMT), FPGA
- Speed: cycle vs. frequency
  - cycle time = 1/(clock frequency)
  - 2.0 GHz = 0.5 ns per cycle
  - CPU Time = Σ(number of clock cycles)/frequency
- Memory: latency vs. bandwidth
  - latency: the time interval between the request for information and the access (to the first bit of that information)
  - Bandwidth: the number of bits per second
- Throughput vs. locality: CPI, MIPS, FLOPS, FMO
- Pipelining: instruction throughput, data dependency, ILP, …

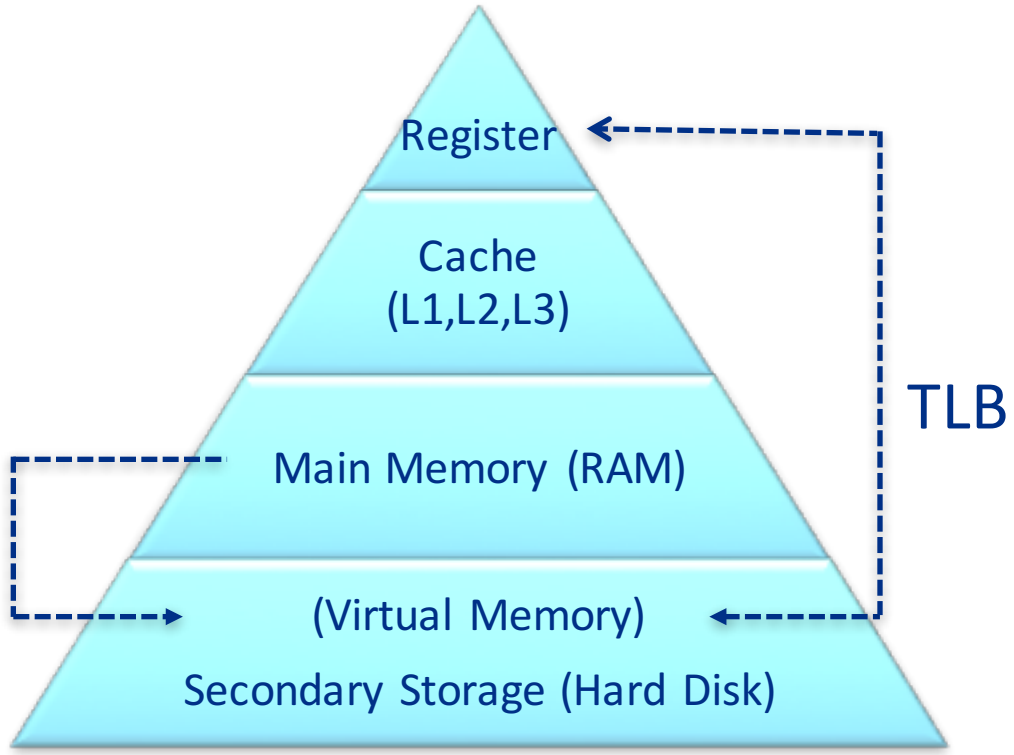
Understanding CPU Performance

- Q1: Which operation takes more cycles?
  1. Integer division
  2. Double division
  3. Function call
  4. static_cast<int>(double)

- Strategies
  - Do not mix data type
  - Avoid unnecessary divisions and function calls in the inner most loop

Understanding Memory Transaction

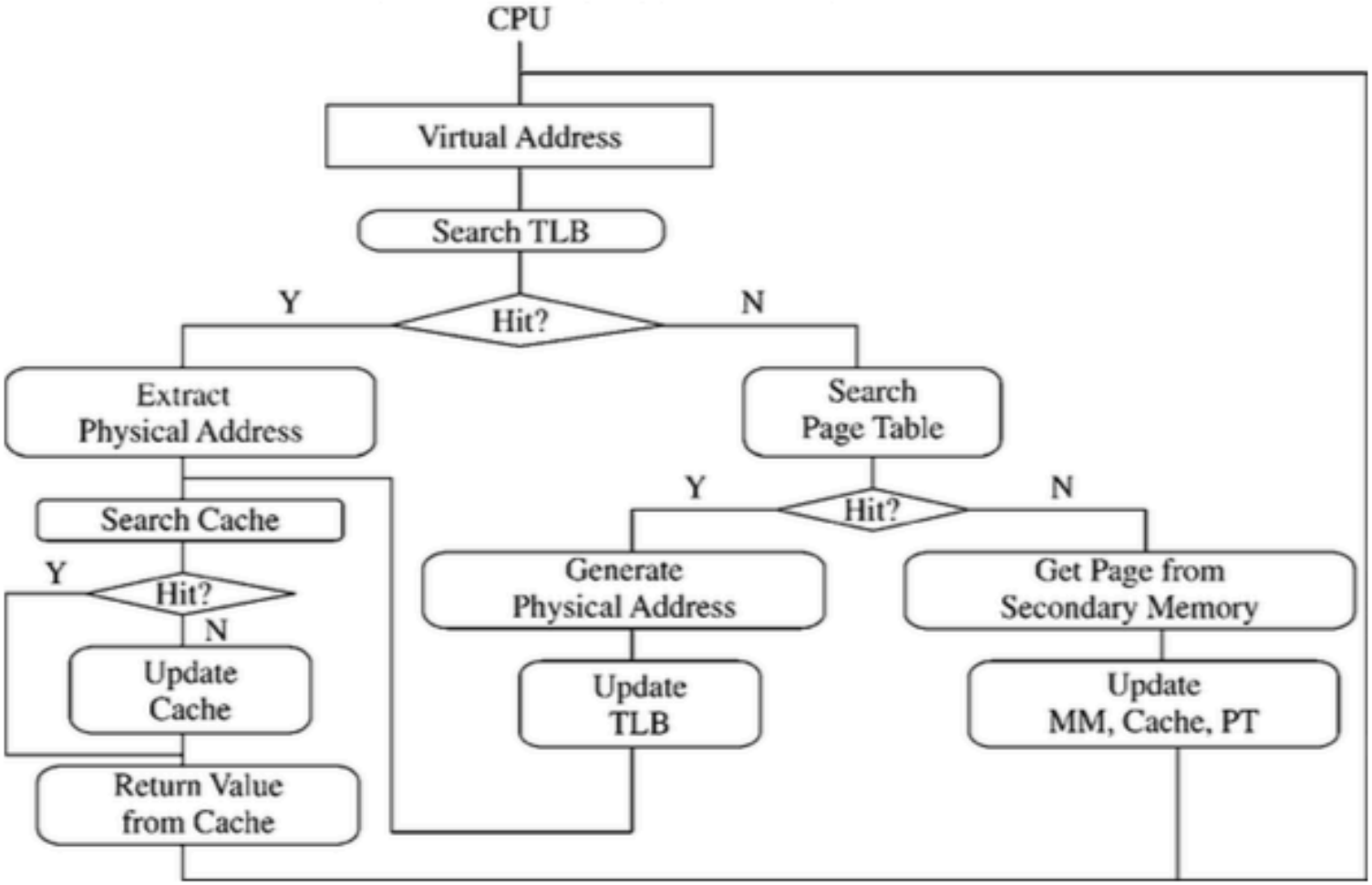
- Memory hierarchy
  - Registers
  - Cache (L1/L2)
  - DRAM (rss)
  - Virtual (vsize)
  - Secondary storage



- Caching
  - Spatial locality (data storage, coalescence)
  - Temporal locality (data reusability in near future)
  - Replacement polices
  - TLB (translation look-aside buffer, the most recent page access)

Understanding Memory Transaction

- Example of memory accesses scenarios



- Do not over-optimize by yourself, but rely on profiling first

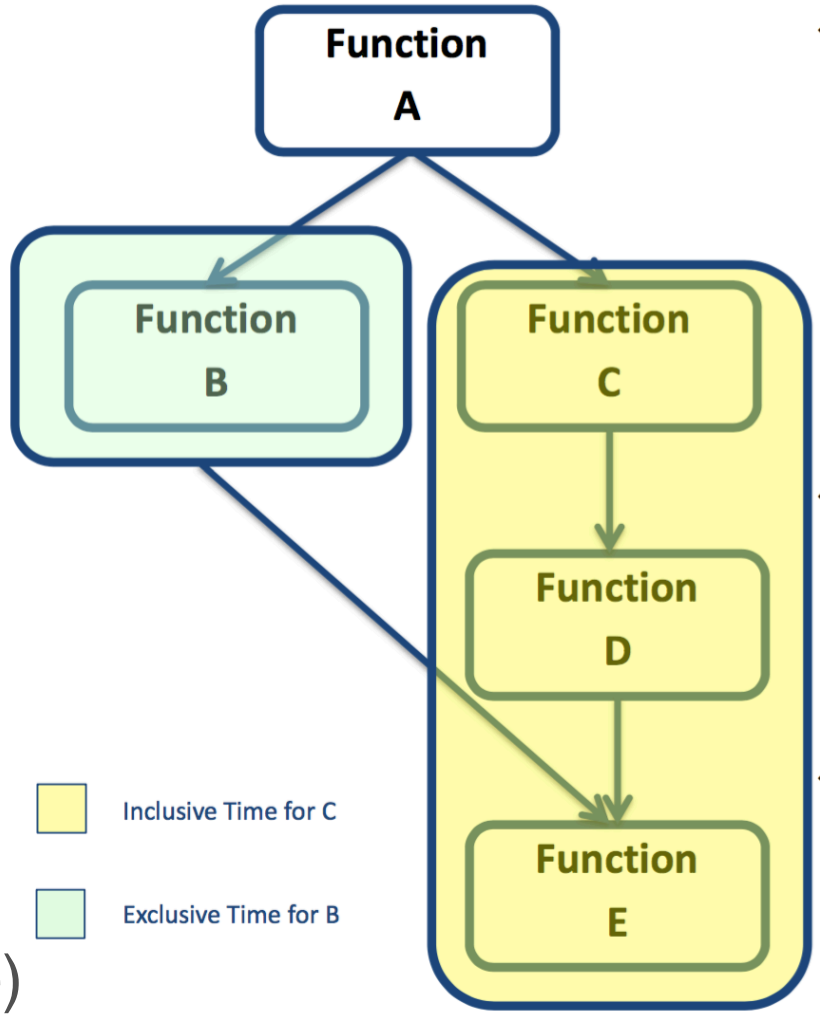
Understanding Memory Performance

- Q2: Which ratio is the biggest in memory access?
  1. L1 Cache/Register
  2. L2 Cache/L1 Cache
  3. RAM/L2 Cache
  4. Virtual Memory/RAM

- Strategies
  - Try to fit everything in RAM
  - Try to fit essential calculations in cache

Basic Concepts of Performance Profilers

- Program segments:
  - Code
  - Stack (program)
  - Heap
- Collecting program events
  - Hardware interrupts
  - Code instrumentation
  - Instruction set simulation
  - Tracing (when)
- Periodic sampling
  - Top of the stack (exclusive)
  - Anywhere in the stack (inclusive)



Classification of Profilers by Techniques used

- Instrumentation: inserts extra code at each function call to count how many times the function is called and how much time it takes.
- Sampling: The profiler tells the operating system to generate an interrupt and counts how many times an interrupt occurs in each part of the program
  - no modification of the program
  - time-based
  - event-based
- Debugging tools: The profiler inserts temporary debug breakpoints at every function or every code line (valgrind)

Examples of Profilers

- Basic OS tools:
  - gprop/perf
  - cachegrind/callgrind
- Hardware counter
  - PAPI and tools set
- Vendor tools
  - Intel VTune Amplifier XE, Inspector, Advisor, ITAC
  - AMD CodeAnalyst
  - Allinea (map and DDD)
- ASCR tools (Open source)
  - HPCToolkit (Rice Univ.)
  - TAU (Oregon Univ.)
  - OpenISpeedshop (Krell)
- HEP
  - FAST (FNAL)
  - IgProf
  - Gooda

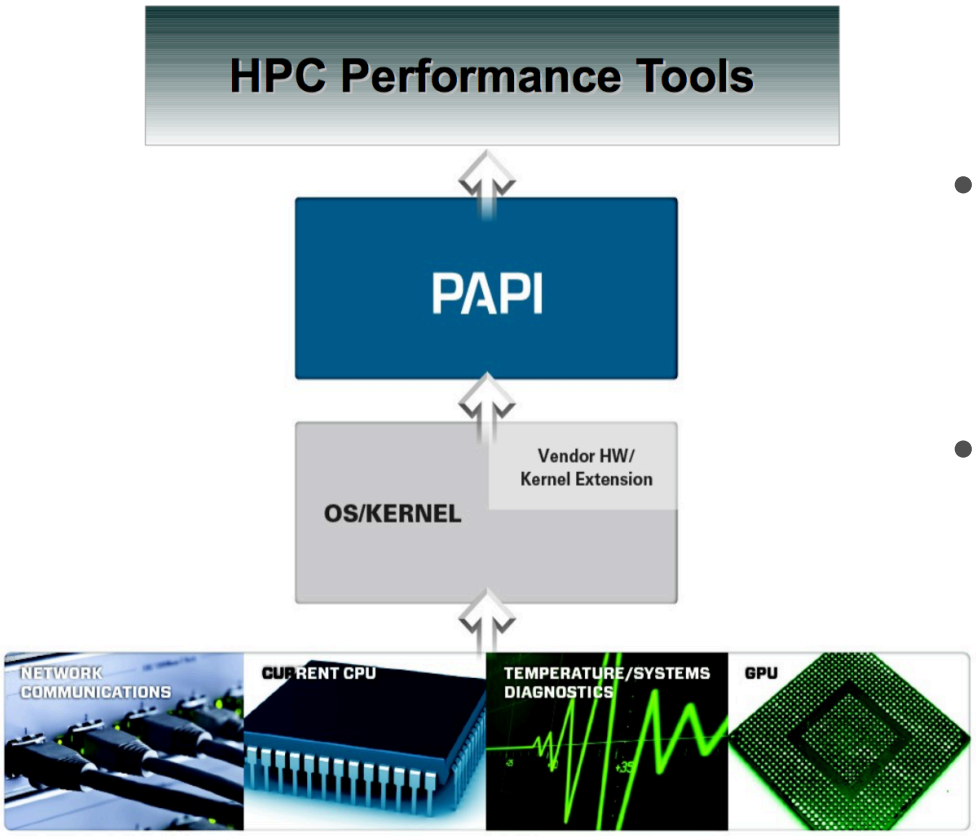
gprof: demo

- Compile your program with gcc using –pg flag
- Run your program (as usual) – will produce gmon.out
- Run gprof

- **wget https://g4cpt.fnal.gov/g4p/demos/demo.cc**
- **g++ -pg demo.cc -o demo**
- **time ./demo**
- **gprof ./demo**

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 36.49      0.51     0.51        1   514.51   514.51  Function_C()
 34.34      1.00     0.48        1   484.25   998.76  Function_B()
 30.05      1.42     0.42        1   423.72   423.72  Function_A()
  0.00      1.42     0.00        1     0.00     0.00  global constructors keyed to _Z10Function_Cv
  0.00      1.42     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int,   int)
```

PAPI (Performance API)



- A standard API to access hardware performance counters
- Relation between software performance and processor events
- Event metrics
  - FLOPS, Load/Store
  - cache hit/miss, TLB miss
  - power consumption (MuMMI)
  - platform specific metrics

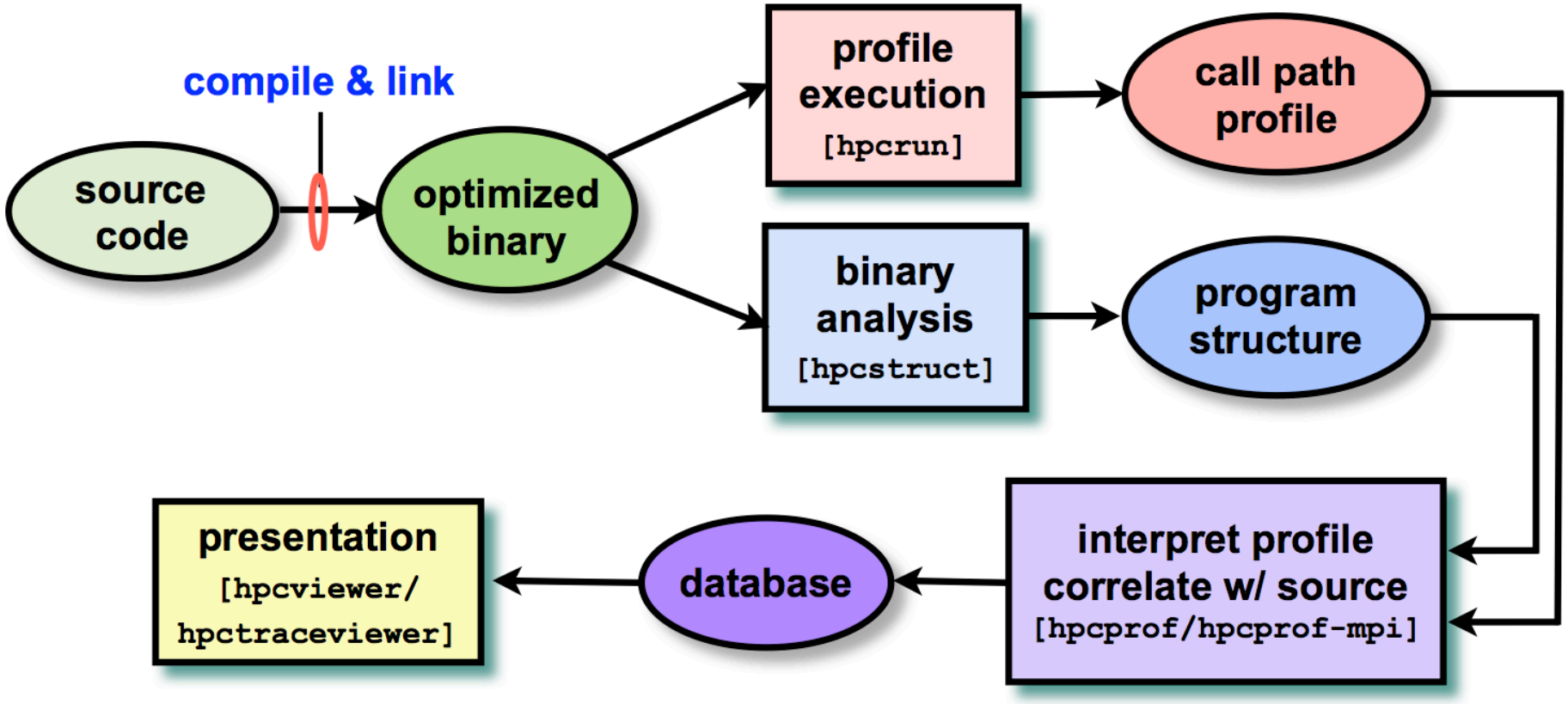
Hardware Counters

- Operating systems support both non-derived and derived PAPI presets: `papi_avail -a` for listing
- A list of some possible hardware counter combinations

For **Xeon** processors:	
PAPI_FP_INS, PAPI_LD_INS, PAPI_SR_INS	Load store info, memory bandwidth needs
PAPI_L1_DCM, PAPI_L1_TCA	L1 cache hit/miss ratios
PAPI_L2_DCM, PAPI_L2_TCA	L2 cache hit/miss ratios
LAST_LEVEL_CACHE_MISSES, LAST_LEVEL_CACHE_REFERENCES	L3 cache info
MEM_UNCORE_RETIRED:REMOTE_DRAM, MEM_UNCORE_RETIRED:LOCAL_DRAM	Local/nonlocal memory access
For **Opteron** processors:	
PAPI_FAD_INS, PAPI_FML_INS	Floating point add multiply
PAPI_FDV_INS, PAPI_FSQ_INS	Square root and divisions
PAPI_FP_OPS, PAPI_VEC_INS	Floating point and vector instructions
READ_REQUEST_TO_L3_CACHE:ALL_CORES, L3_CACHE_MISSES:ALL_CORES	L3 cache

Example of Sampling Tools and Workflow: HPCToolkit

- Typically unmodified binary and call stack analysis
- Code centric view, GUI and text-based flat profile



Example of Integrated Tools: TAU (Tuning Analysis Utilities)

- Dynamic, compiler based, source based Instrumentation
- Analysis tools
  - ParaProf
  - PerfExplorer
  - Tracer (Jumpshot, vampir)
- Various built-in graphical tools
- Dis/advantage: compiler/source-based instrumentation

Understanding Sampling Profilers

- Q3: What are the disadvantages of sampling profilers?
  1. Sampling uncertainty (→ statistical analysis with repetition)
  2. Non-Reproducibility (→ use definitive tools)
  3. Interference from other processes (→ standalone nodes)
  4. Jumping between cores (→ setting NUMA affinity, pinning)
  5. All of above

- Strategies
  - Understand your program first (intensity: arithmetic vs. memory)
  - Overview with sampling experiments
  - Focus on critical parts of code: Rule of 80:20
  - Detailed optimization with hardware counter experiments
  - Benchmarking and monitoring of every minor/major update

Fermilab U.S. DEPARTMENT OF ENERGY | Office of Science

PART-II

Profiling Results of LArTest with IgProf and OpenlSpeedshop

Application for this Tutorial: LArTest

- A standalone Geant4 application (developed by H. Wenzel)
  - Cubic (5mx5mx5m) LAr fiducial volume
  - GDML to assign step limits and sensitive detector to volumes
  - Optical (scintillation) photons produced in sensitive detector
- Computing performance monitoring features
  - Event time
  - Memory (IgProf, statm)
  - Statistics of the number for tracks/steps per particle type
- Profiling examples with IgProf and OpenlSpeedshop

Installing and Running IgProf

- Installation: http://igprof.org/install.html
- Implementation

```
if (void *sym = dlsym(0, "igprof_dump_now")) {
   dump_ = __extension__ (void(*)(const char *)) sym;
} else { /* message */ ; }
```

- Running igprof on your application (-mp: memory profiling) http://igprof.org/running.html

```
igprof -d -mp -z -o ${IG_OUT} $exe {args...}
```

- Analysis (web-navigable version of the report, -r for $mode)

```
cmd="igprof-analyse --sqlite -d -v -g –r"
$cmd ${mode} ${IG_OUT} | sqlite3 out.sql3
```

  - ${mode} = MEM_LIVE, MEM_MAX, MEM_TOTAL

IgProf

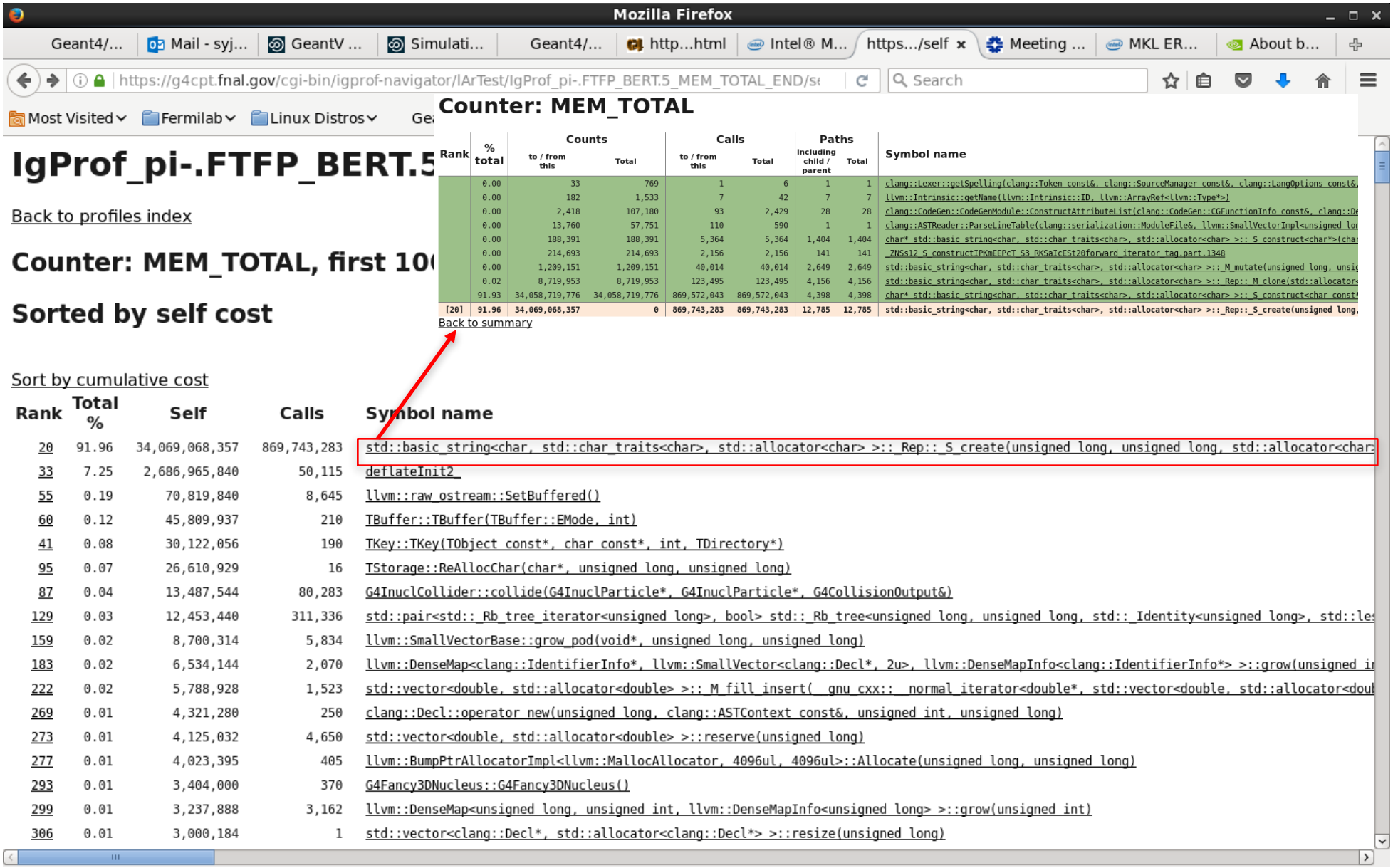
- Snapshot live memory on the heap (for every N-events)

```
if ( dump_ && evt->GetEventID() % 25 == 0) {
   sprintf(outfile,"|gzip -9c > IgProf.%d.gz",evt->GetEventID()+1);
   dump_(outfile);
}
```

```
cmd="igprof-analyse --sqlite -d -v -g -r"
$cmd ${mode} -b out1.gz --diff-mode out2.gz| sqlite3 diff.sql3
```

- Performance report formats
  - ascii text (flat file)
  - sqlite database files
- Demo for the web-navigable report
  https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_igprof.html

LArTest: IgProf (TOTAL MEM)



Rank	Total %	Self	Calls	Symbol name
20	91.96	34,069,068,357	869,743,283	std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_S_create(unsigned long, unsigned long, std::allocator<char>
33	7.25	2,686,965,840	50,115	deflateInit2_
55	0.19	70,819,840	8,645	llvm::raw_ostream::SetBuffered()
60	0.12	45,809,937	210	TBuffer::TBuffer(TBuffer::EMode, int)
41	0.08	30,122,056	190	TKey::TKey(TObject const*, char const*, int, TDirectory*)
95	0.07	26,610,929	16	TStorage::ReAllocChar(char*, unsigned long, unsigned long)
87	0.04	13,487,544	80,283	G4InuclCollider::collide(G4InuclParticle*, G4InuclParticle*, G4CollisionOutput&)
129	0.03	12,453,440	311,336	std::pair<std::_Rb_tree_iterator<unsigned long>, bool> std::_Rb_tree<unsigned long, unsigned long, std::_Identity<unsigned long>, std::le
159	0.02	8,700,314	5,834	llvm::SmallVectorBase::grow_pod(void*, unsigned long, unsigned long)
183	0.02	6,534,144	2,070	llvm::DenseMap<clang::IdentifierInfo*, llvm::SmallVector<clang::Decl*, 2u>, llvm::DenseMapInfo<clang::IdentifierInfo*> >::grow(unsigned ir
222	0.02	5,788,928	1,523	std::vector<double, std::allocator<double> >::_M_fill_insert(__gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<doub
269	0.01	4,321,280	250	clang::Decl::operator new(unsigned long, clang::ASTContext const&, unsigned int, unsigned long)
273	0.01	4,125,032	4,650	std::vector<double, std::allocator<double> >::reserve(unsigned long)
277	0.01	4,023,395	405	llvm::BumpPtrAllocatorImpl<llvm::MallocAllocator, 4096ul, 4096ul>::Allocate(unsigned long, unsigned long)
293	0.01	3,404,000	370	G4Fancy3DNucleus::G4Fancy3DNucleus()
299	0.01	3,237,888	3,162	llvm::DenseMap<unsigned long, unsigned int, llvm::DenseMapInfo<unsigned long> >::grow(unsigned int)
306	0.01	3,000,184	1	std::vector<clang::Decl*, std::allocator<clang::Decl*> >::resize(unsigned long)

Rank	% total	Counts: to / from this	Counts: Total	Calls: to / from this	Calls: Total	Paths: Including child / parent	Paths: Total	Symbol name
	0.00	33	769	1	6	1	1	clang::Lexer::getSpelling(clang::Token const&, clang::SourceManager const&, clang::LangOptions const&
	0.00	182	1,533	7	42	7	7	llvm::Intrinsic::getName(llvm::Intrinsic::ID, llvm::ArrayRef<llvm::Type*>)
	0.00	2,418	107,180	93	2,429	28	28	clang::CodeGen::CodeGenModule::ConstructAttributeList(clang::CodeGen::CGFunctionInfo const&, clang::De
	0.00	13,760	57,751	110	590	1	1	clang::ASTReader::ParseLineTable(clang::serialization::ModuleFile&, llvm::SmallVectorImpl<unsigned lon
	0.00	188,391	188,391	5,364	5,364	1,404	1,404	char* std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_S_construct<char*>(char
	0.00	214,693	214,693	2,156	2,156	141	141	_ZNSs12_S_constructIPKmEEPcT_S3_RKSaIcESt20forward_iterator_tag.part.1348
	0.00	1,209,151	1,209,151	40,014	40,014	2,649	2,649	std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_mutate(unsigned long, unsig
	0.02	8,719,953	8,719,953	123,495	123,495	4,156	4,156	std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_M_clone(std::allocator<
	91.93	34,058,719,776	34,058,719,776	869,572,043	869,572,043	4,398	4,398	char* std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_S_construct<char const*
[20]	91.96	34,069,068,357	0	869,743,283	869,743,283	12,785	12,785	std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_S_create(unsigned long,

Hint from IgProf

- Q4: How to improve performance of this function? Avoid …

```
MaterialPropertyVector*
MaterialPropertiesTable::GetProperty(const char *key)
{
  // Returns a Material Property Vector corresponding to a key
  if (string(key) == "GROUPVEL") return SetGROUPVEL();

  MPTiterator i;
  i = MPT.find(string(key));
  if ( i != MPT.end() ) return i->second;
  return nullptr;
}
```

1. String comparison
2. String conversion
3. String search (find)
4. Race condition (map)

Introduction to OpenlSpeedshop (OSS)

- Comprehensive performance analysis of sequential, multithreaded, and MPI applications
- Open source (the Krell institute, https://openspeedshop.org) and one of ASCR profiling tools
- The base functionality includes
  - Sampling experiment (light-weighted)
  - Support call stack analysis
  - Hardware performance (PAPI) counters
  - Multi-threaded, MPI profiling and tracing
  - Memory function tracing, I/O profiling and tracing, etc…
- Tested on a variety of Linux clusters and supports parallel hardware architectures (Intel MIC, NVIDIA CUDA) as well as HPC systems (Cray, Blue Gene)

OSS: Installation and Performance Measurement

- Installation: a typical build (with the version 2.2)

```
./install-tool --build-krell-root
               --krell-root-prefix  ${install_dir}/krellroot_v2.2
               --with-openmpi /usr/local/openmpi-1.8.1
./install-tool --build-offline
               --openss-prefix ${install_dir}/openspeedshop2.2
               --krell-root-prefix ${install_dir}/krellroot_v2.2
               --with-openmpi /usr/local/openmpi-1.8.1
```

- Running an experiment: unmodified binary instrumentation

  osspcsamp "lArTest lArBox.gdml profile.pi-5GeV" [frequency]

- Performance analysis: command-line (-cli) or GUI (-f)

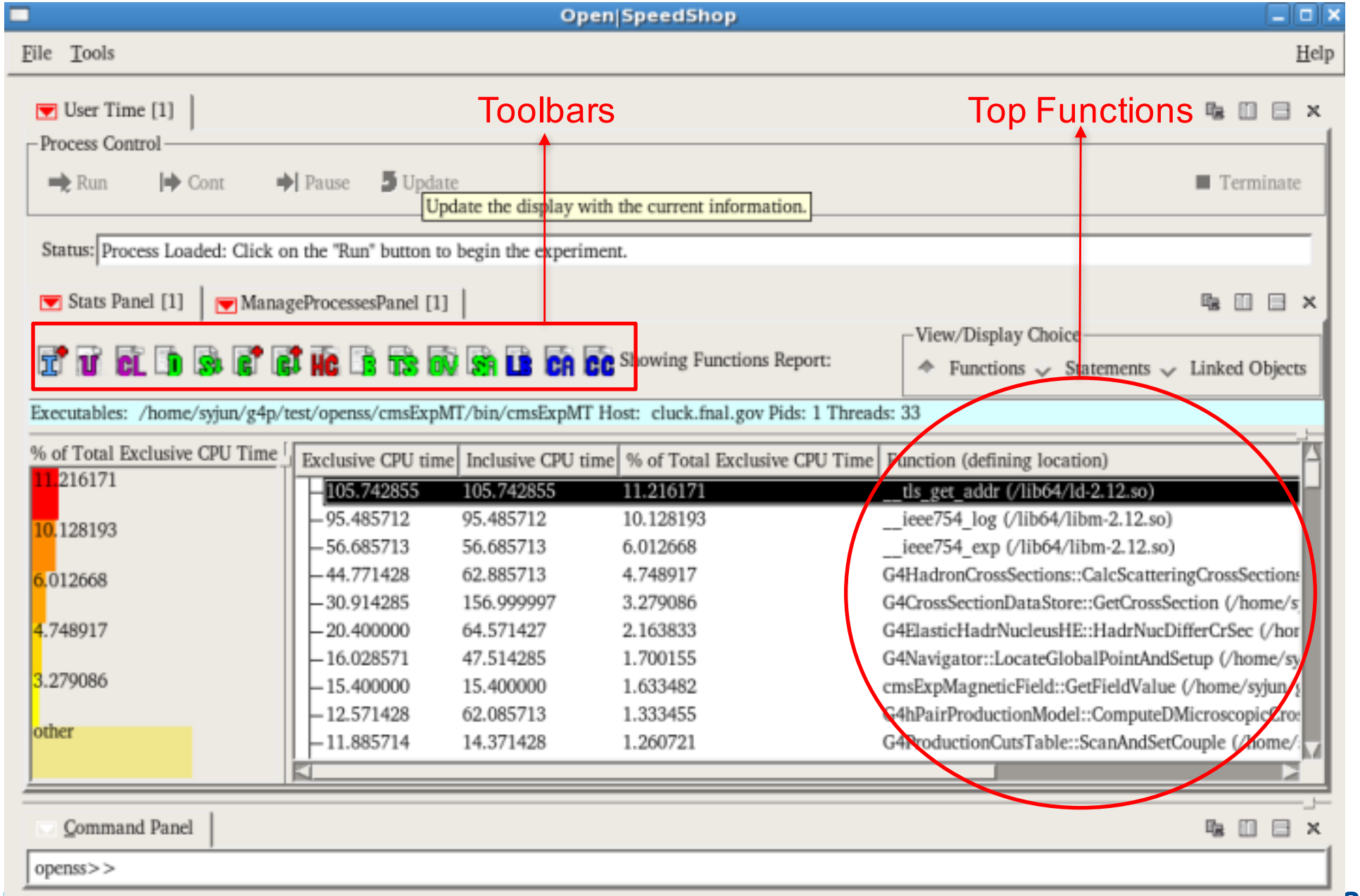
  openss -cli lArTest-pcsamp.openss

Demo: OSS Command-line Analysis (-cli)

> openss –cli	Open the CLI.
openss>> expcreate –f "mutatee 2000" pcsamp	Create an experiment using pcsamp with this application.
openss>> expgo	Run the experiment and create the database
openss>> expview	Display the default view of the performance data.

help or help commands	list –v obj
expview	list –v ranks
expview -v statements	list –v hosts
expview -v loops	expview –m <metric>
expview -v linkedobjects	expview –v calltrees,fullstack <experiment type> <number>
expview -v calltrees,fullstack	
expview -m loadbalance	expview –v calltrees,fullstack usertime2
expview –r <rank_num>	expview <experiment-name><number>
expcompare –r 1 –r 2 –m time	expview pcsamp2
list –v metrics	expview –v statements <experiment-name><number>
list –v src	

OSS (GUI): Default View and Statistical Panel



Sampling Experiments in OSS

- pcsamp (periodic sampling of program counters)
  - low overhead overview of time distribution
- usertime (call path profiling)
  - inclusive and exclusive timing data
  - call paths, caller and callee relationships
- hwcsamp (periodic sampling hardware counters)
  - profiling of hardware counter events (PAPI events)
- pthreads (POSIX thread tracing)
- mem (memory tracing)
  - call paths of memory related function call events
  - aggregate and individual rank, thread, or processing times
- io (I/O tracing)
- Many other useful experiments

OSS: Measurement Overheads and Output Size

- pcsamp: exclusive time - insensitive to sampling frequency (default 100Hz)

Frequency	Time(sec)	OverHead(%)	DB size(MB)
base :	52.20	-	
50 Hz:	52.27	0.13	0.376832
100 Hz:	52.62	0.80	0.486400
200 Hz	52.36	0.31	0.607232
500 Hz	52.98	1.49	0.811008
1000 Hz:	52.65	0.86	0.971776
10000 Hz:	52.76	1.07	1.012736

- usertime: inclusive time and call paths – large overhead (default 35): similar overhead for hwcsamp

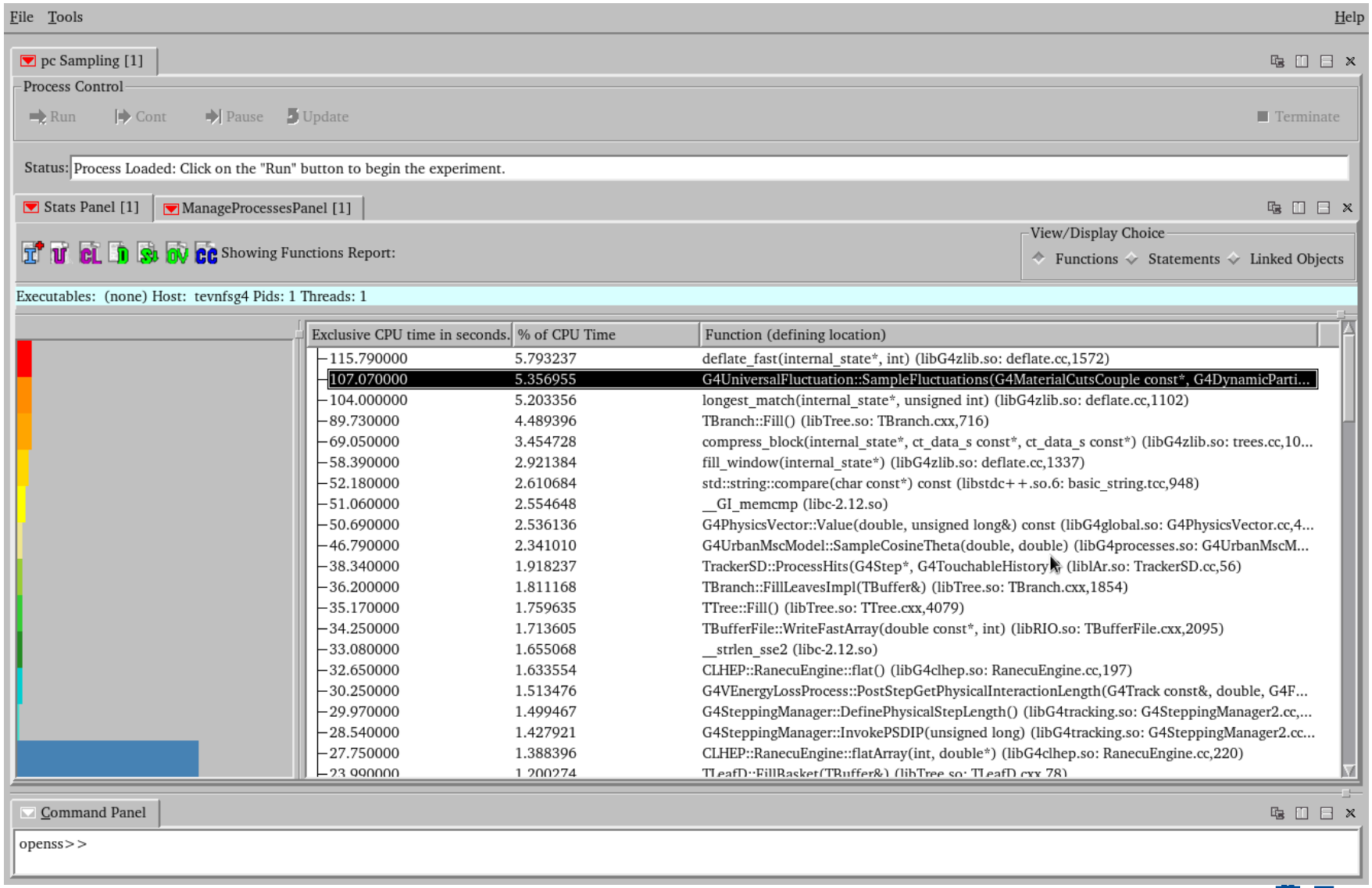
Frequency	Time(sec)	OverHead(%)	DB size(MB)
base :	52.80	-	
35 Hz:	53.89	2.06	1.087488
50 Hz:	54.33	2.90	1.430528
100 Hz:	56.21	6.46	2.355200
200 Hz	60.25	14.11	4.208640
1000 Hz:	92.84	75.83	18.725888

Preliminary Performance Experiments with LArTest

- LArTest configuration
  - Beam: 5 GeV pi-
  - Step limit: 0.01 cm
  - Physics list: FTFP_BERT (uses standard EM)
  - 1000 events
- osspcsamp (100 Hz)
  - I/O (digitization) ON
  - Analysis ON
- ossusertime and osshwcsamp (35 Hz)
  - I/O (digitization) OFF
  - Analysis OFF

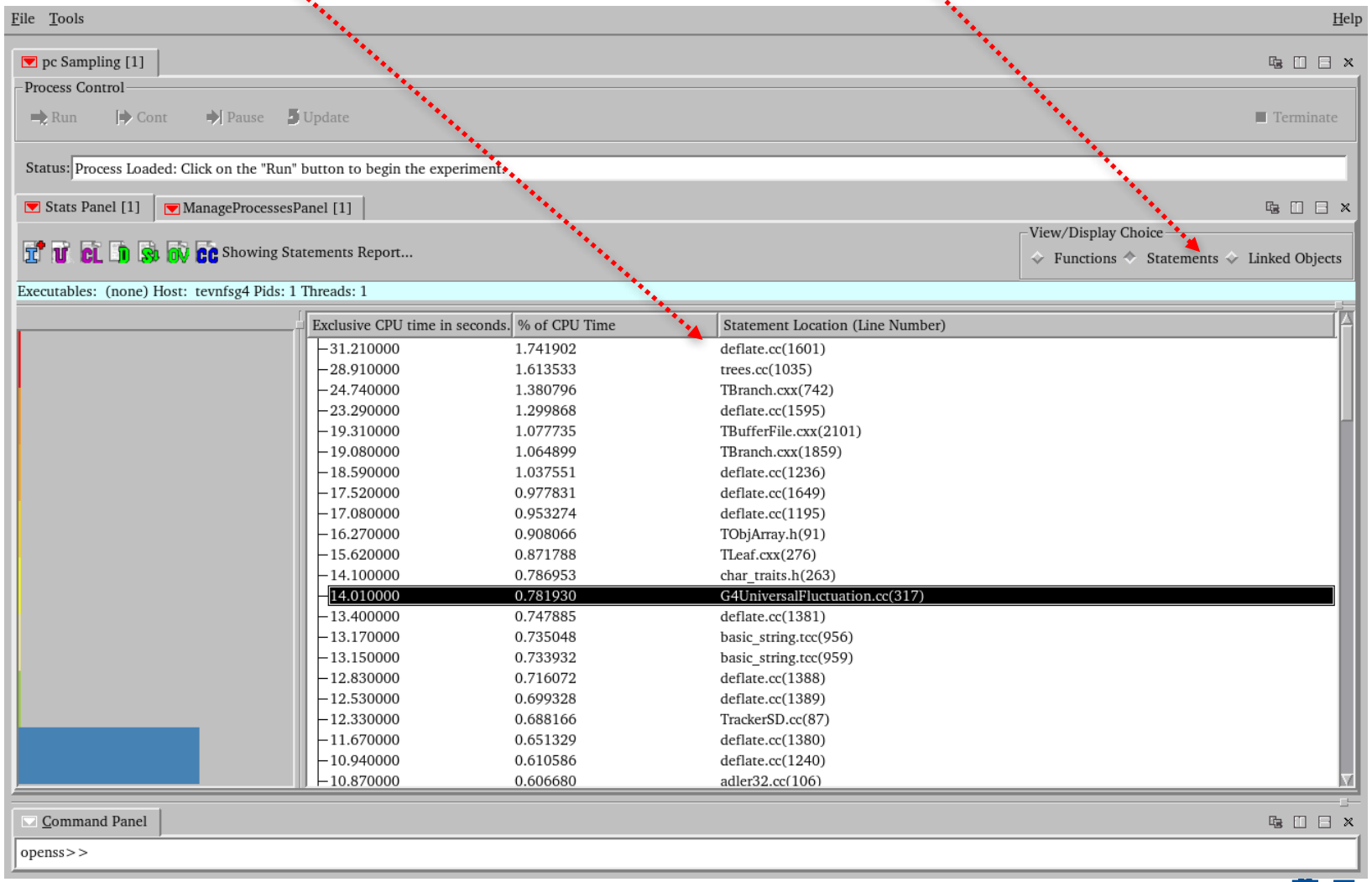
osspcsamp : Functions

- Exclusive CPU time - an overall performance view



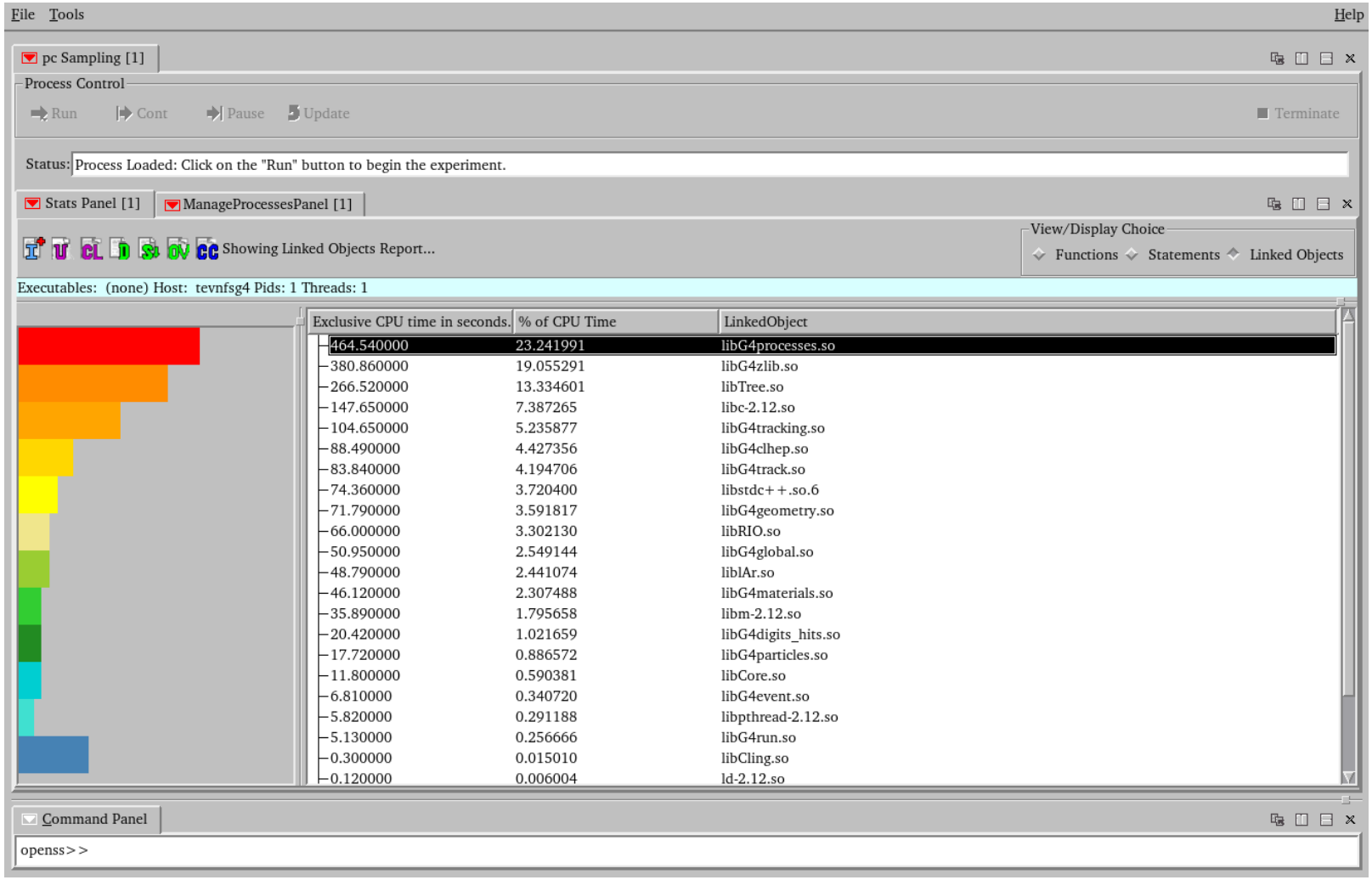
osspcsamp: Statements (Line Numbers)

- Select statement level granularity
- List line numbers in program that took most of time



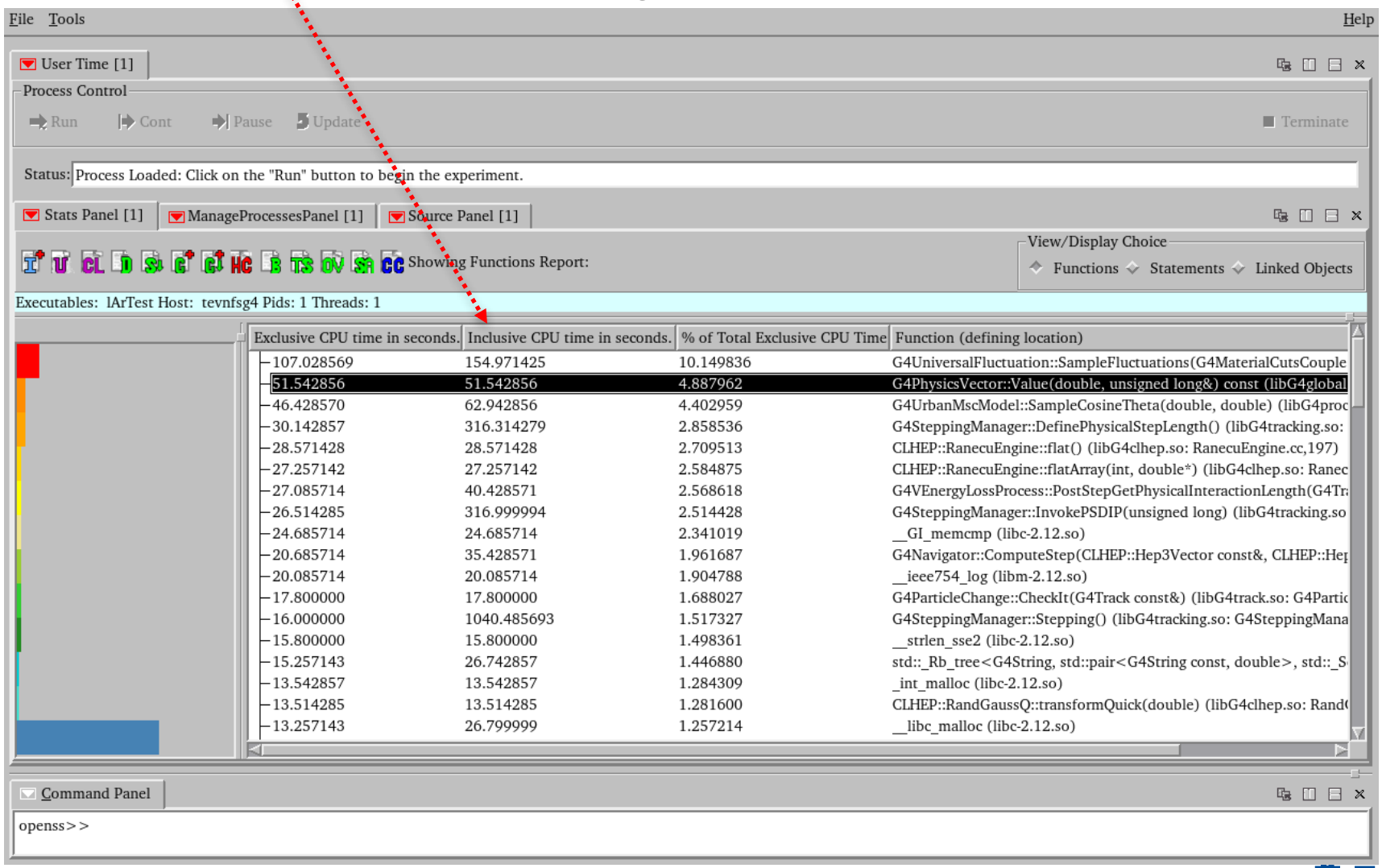
osspcsamp: Linked Objects

- The library in which the associated function is located (aggregated by shared objects)



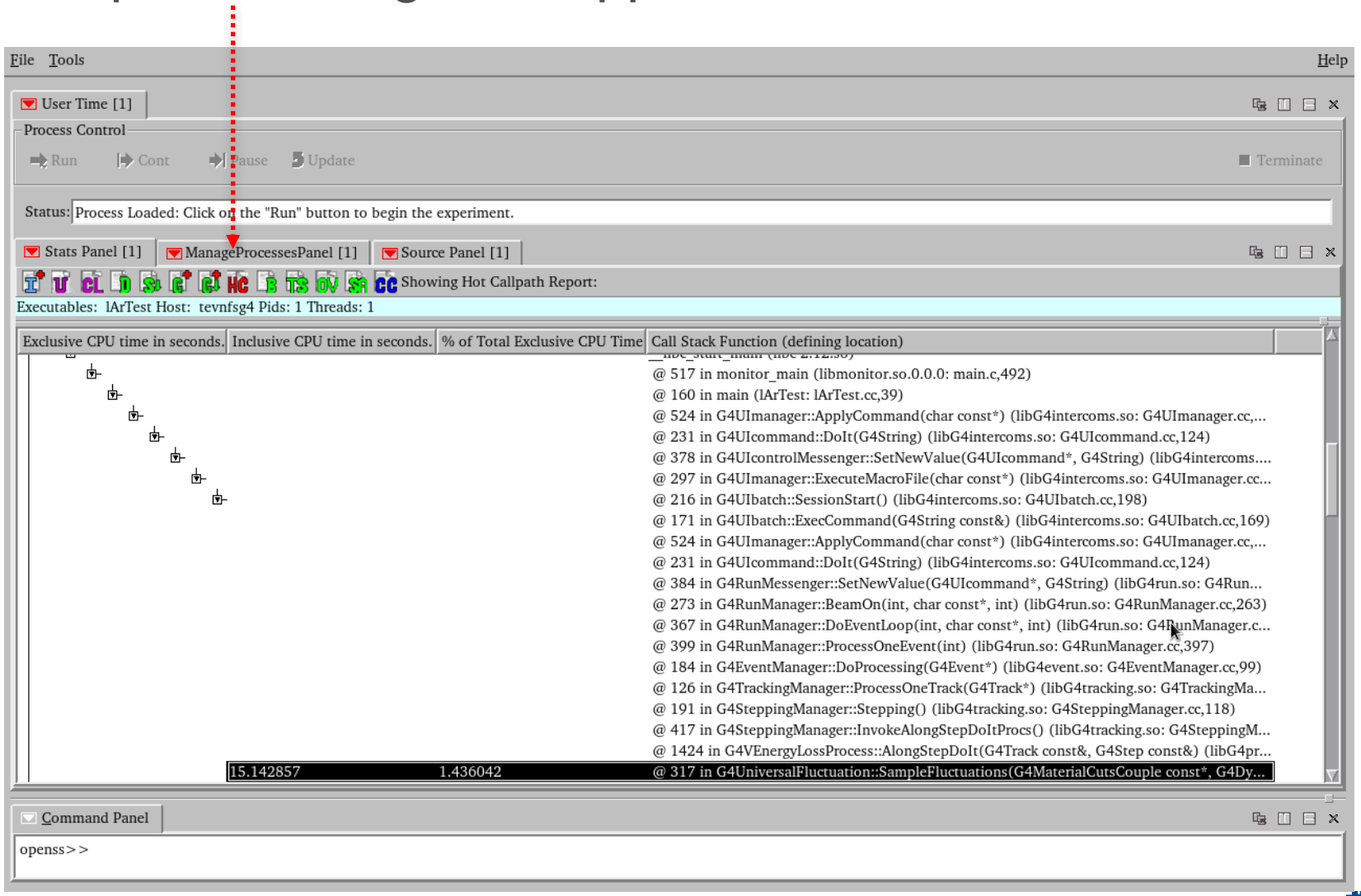
ossusertime: Call Path (Functions)

- Function calls observed anywhere in the stack
- The inclusive time taken by the function and all its callees



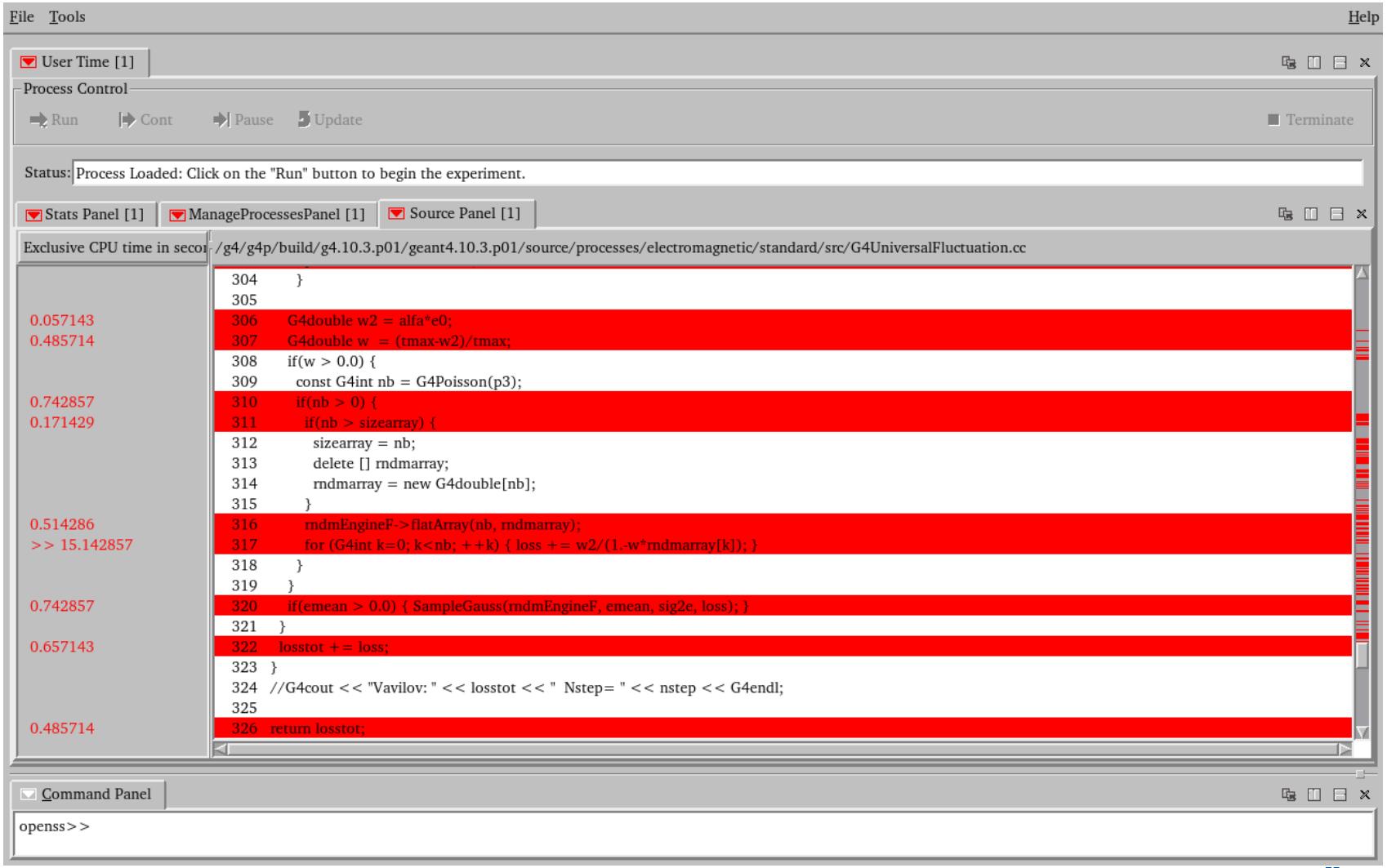
ossusertime: Hot Call Path

- Relationship between caller and callee
- The paths through the application that take the most time



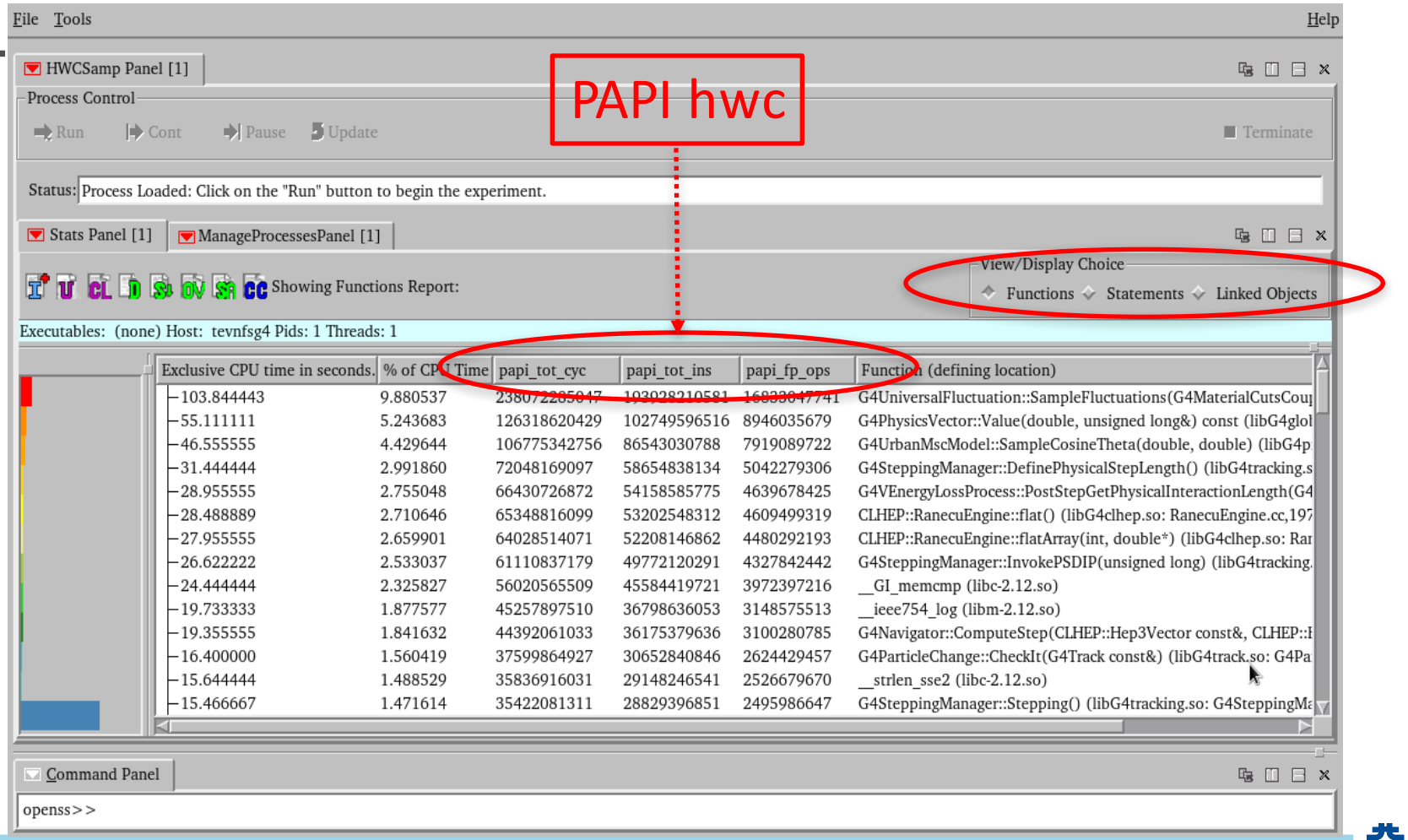
ossusertime: Hot Call (Source)

- Exclusive time on highlighted lines that indicate relatively high CPU times



Experiments with Hardware Counters

- Periodic sampling of hardware counters (hwcsamp)
- Supports both derived and non-derived PAPI presets
- Metrics for instructions, FLOPS, memory and resource patterns



Fermilab

Code Performance by Hardware Counter Metrics

- Derivatives: examples

Hardware Counter Metrics Derivatives	Performance
IPC (Instruction/Cycle)	Large values suggest good balance with minimal stalls.
FPC (FLOPS/Cycle)	Large values for floating point intensive code suggests efficient CPU utilization
FMO (FLOPS/Memory Ops)	Good data locality, Computational Intensity
LPC (Loads/Cycle)	Useful for calculating FMO, may indicate good stride through arrays.
SPC (Stores/Cycle)	Useful for calculating FMO, may indicate good stride through arrays.

- LArTest (Overall): 5 GeV pi- (Intel Xeon X5650@2.67GHz)
  - IPC = 0.79 (relatively small)
  - FMO = 0.32

Other useful OSS Features

- Flexible analysis options (GUI, command line, online)
- Export report data in different formats (text, cvs, chart)
- Multi-threading capability
- Compare two experiments (osscompare): examples
  - two releases
  - two experiments with the different numbers of threads
- Call path analysis based on DB
- Experiments for parallel code (MPI tracing)

Demo: Example of Performance Profiling Report

- Monitor Geant4 part of performance changes for LAr-based detectors by
  - Beam energy/Particle type/Physics list
  - Geant4 (reference) release

https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_sprof.html

https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_igprof.html

OpenlSpeedshop

Geant4.10.3.r04 lArTest

Sample	Physics List	B-Field	Energy
e-	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
e+	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
mu-	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
mu+	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
pi-	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
pi+	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
K-	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
K+	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV
p	FTFP_BERT	OFF (0 T)	1 GeV 5 GeV

Memory Profiler/IgProf

Memory profiling reports

- MEM_LIVE: memory that has not been freed - snapshot of the heap, i.e. a heap profile.
- MEM_MAX: the largest single allocation by any function
- MEM_TOTAL: the total amount of memory allocated by any function - a snapshot of poor memory locality

- N: memory snapshot at the end of N-th event
- Diff(N-M): memory difference between N-th and M-th event - direct memory leakage
- End of Run: memory snapshot at the End of Run

Geant4.10.3.r04 lArTest B=4.0T

Sample	Physics List	Energy	MEM_LIVE	MEM_MAX	MEM_TOTAL
e-	FTFP_BERT	1 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
		5 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
e+	FTFP_BERT	1 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
		5 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
mu-	FTFP_BERT	1 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
		5 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run

Summary

- Performance profiling and analysis is an essential part of the software development cycle
  - Modern hardware architectures are demanding (parallelism)
  - HEP applications are big and complex
  - Profilers will helps to identify critical parts of code, monitor changes of performance and provide opportunities of optimization
- Where you can start:
  - Try profiling your programs with basic tools
  - IgProf: http://igprof.org/index.html
  - OpenISpeedshop: https://openspeedshop.org/
  - HPCToolkits: http://hpctoolkit.org/index.html
  - TAU: http://www.cs.uoregon.edu/research/tau/home.php
- Above tools are quite suitable for continuous integration tests

Acronym

- CISC: Complicated instruction set computer)
- RISC: Reduced instruction set computer)
- ARM: Advanced RISC Machines
- CPI: Cycles per instruction
- IPC: Instructions per cycle
- MIPS: Million instructions per second
- FMO: Floating point operations per memory operation
- DRAM: Dynamic random-access memory
- ASCR: Advanced Scientific Computing Research
- MuMMI: Multiple Metrics Modeling Infrastructure

Answer sheet

- Q1:
- Q2:
- Q3:
- Q4: