HPCToolkit Components for Measurement, Analysis and Presentation

John Mellor-Crummey, Mark Krentel, Laksono Adhianto

Mike Fagan, Gabriel Marin, Nathan Tallent

Department of Computer Science
Rice University

http://www.hipersoft.rice.edu/hpctoolkit

RICE

Outline

- **Brief overview of the HPCToolkit toolchain**
- **Three new components**
  - — libmonitor
  - — call stack sampling
  - — hpcviewer
- **New and notable**
- **Components**

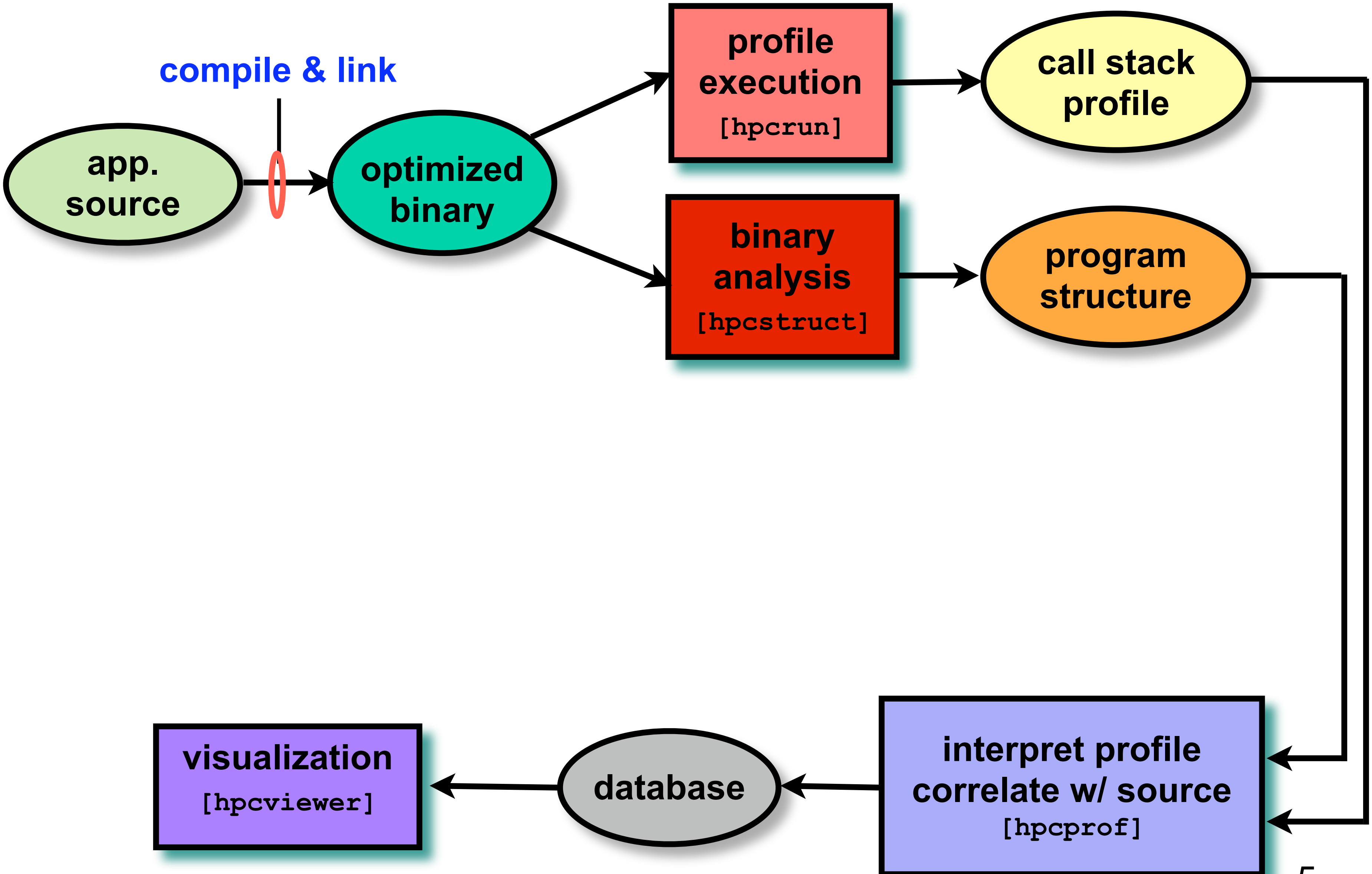
Performance Analysis Goals

- **Accurate measurement of complex parallel codes**
  - — large, multi-lingual programs
  - — fully optimized code: loop optimization, templates, inlining
  - — binary-only libraries, sometimes partially stripped
  - — complex execution environments
    - – dynamic loading
    - – SPMD parallel codes with threaded node programs
    - – batch jobs
- **Effective performance analysis**
  - — insightful analysis that pinpoints and explains problems
    - – correlate measurements with code (yield actionable results)
    - – intuitive enough for scientists and engineers
    - – detailed enough for compiler writers
- **Scalable to petascale systems**

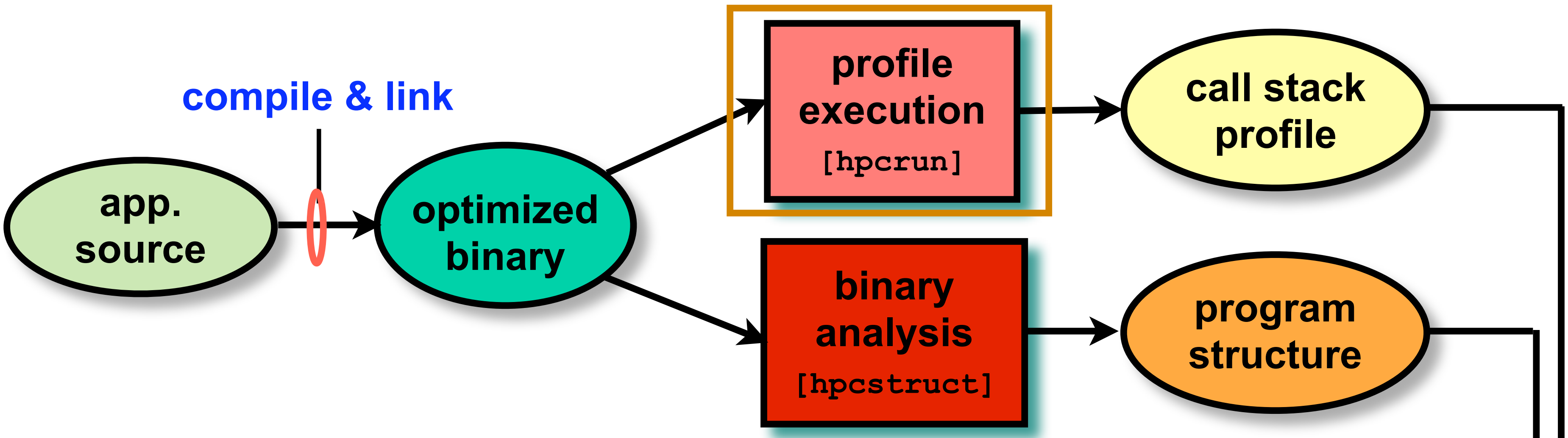
HPCToolkit Design Principles

- Binary-level measurement and analysis
  - — observe fully optimized, dynamically linked executions
  - — support multi-lingual codes with external binary-only libraries
- Sampling-based measurement (avoid instrumentation)
  - — minimize systematic error and avoid blind spots
  - — enable data collection for large-scale parallelism
- Collect and correlate multiple derived performance metrics
  - — diagnosis requires more than one species of metric
  - — derived metrics: “unused bandwidth” rather than “cycles”
- Associate metrics with both static and dynamic context
  - — loop nests, procedures, inlined code, calling context
- Support top-down performance analysis
  - — intuitive enough for scientists and engineers to use
  - — detailed enough to meet the needs of compiler writers

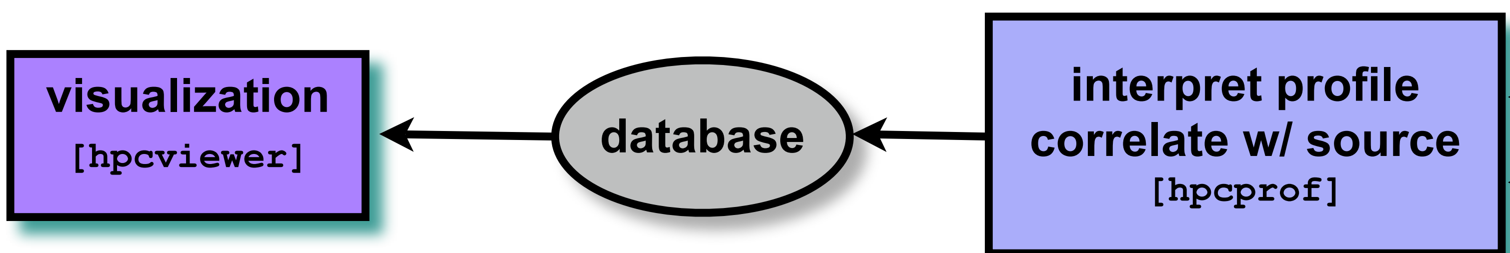
HPCToolkit Workflow



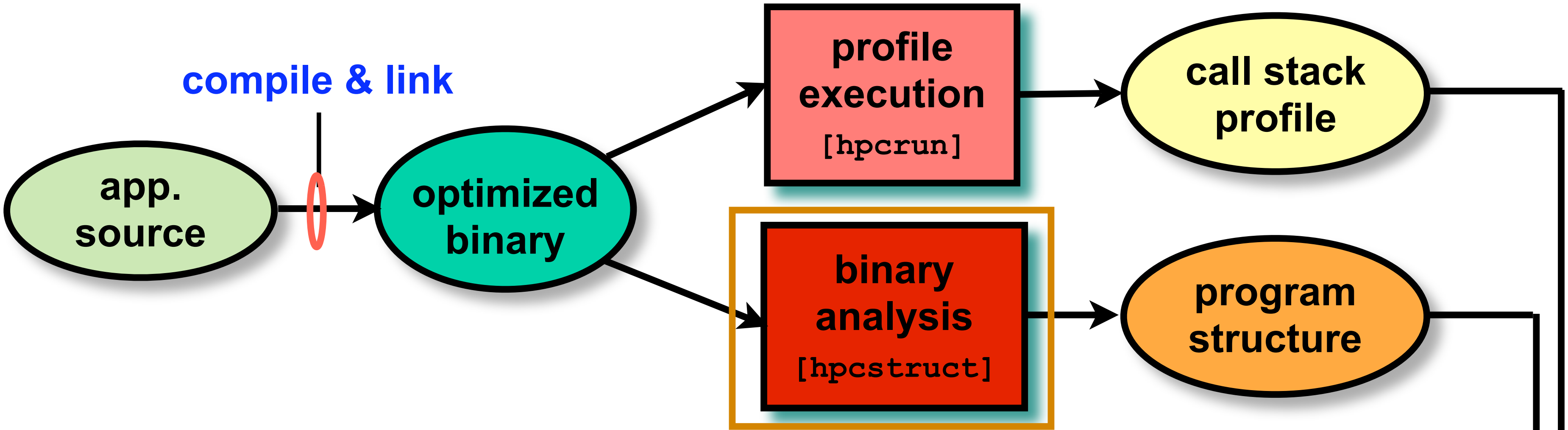
HPCToolkit Workflow



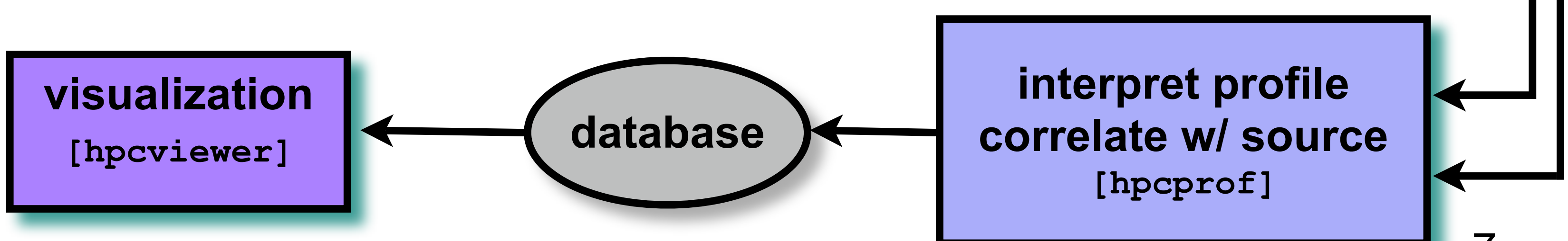
- **Measure execution unobtrusively**
  - launch unmodified, optimized application binaries
  - collect call path (or flat) profiles of events of interest



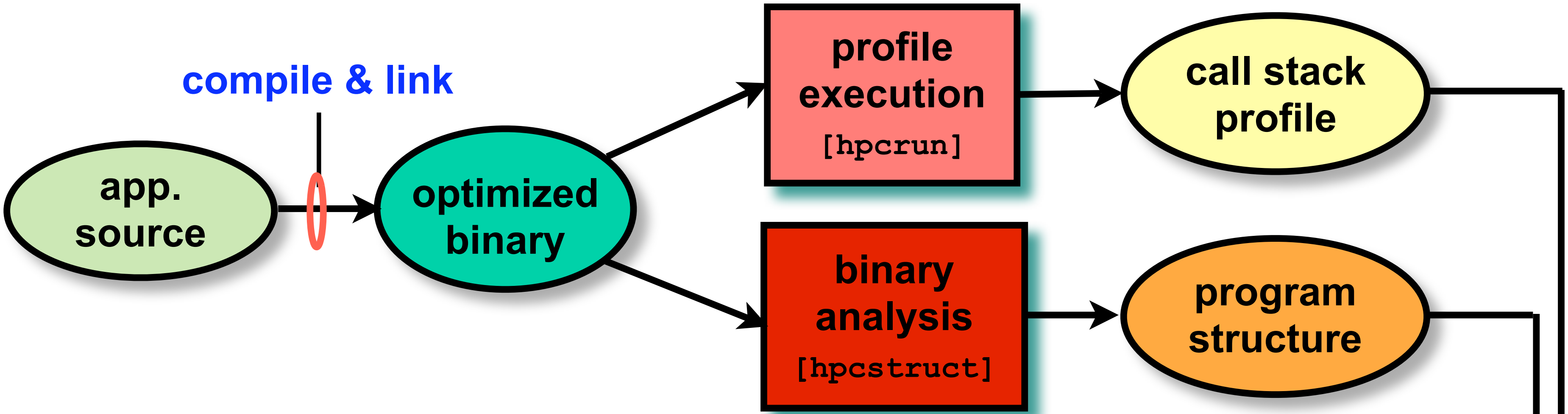
HPCToolkit Workflow



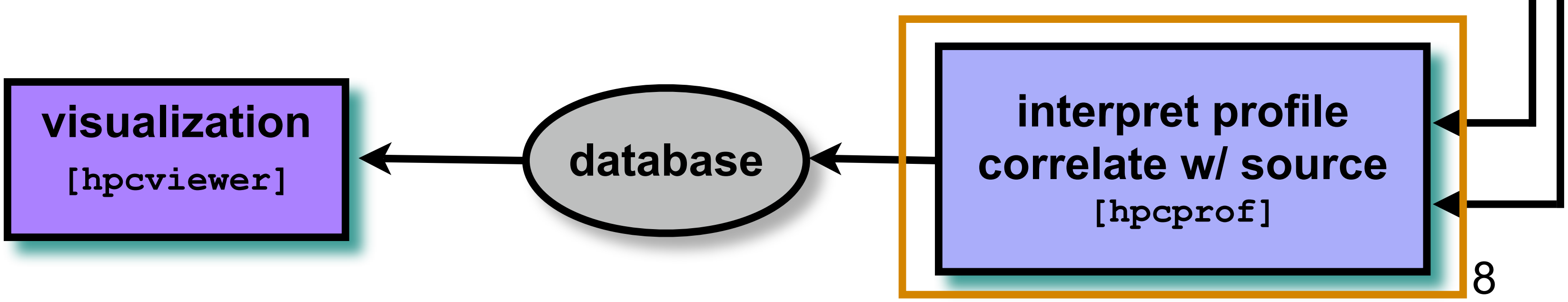
- **Analyze binary to recover program structure**
  - — extract loop nesting & identify procedure inlining
  - — map transformed loops and procedures to source



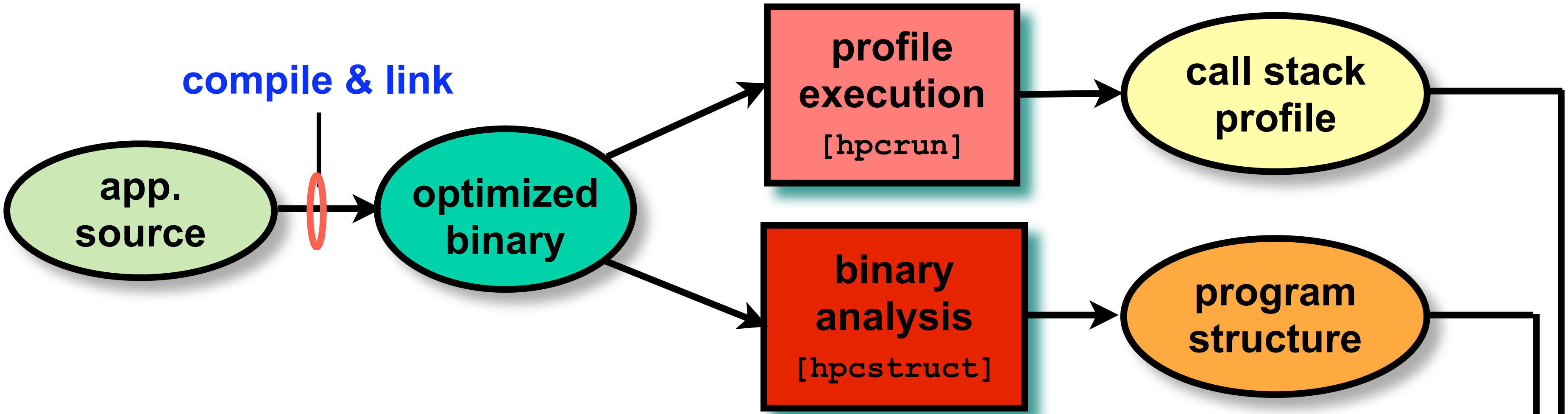
HPCToolkit Workflow



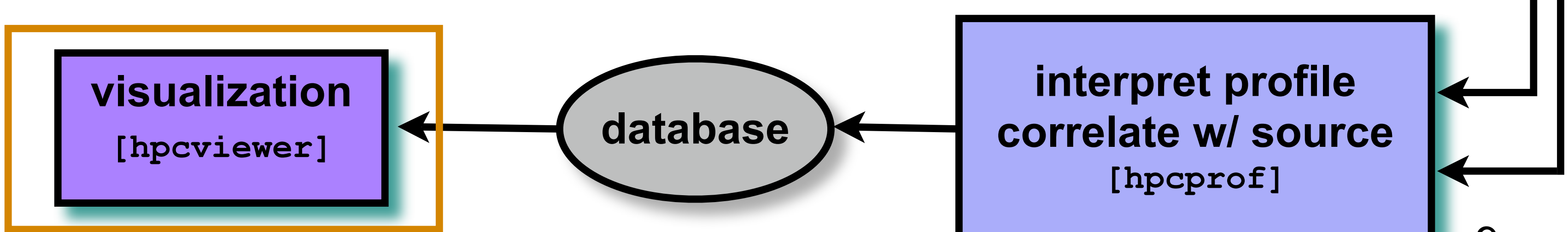
- Combine multiple profiles
- Correlate dynamic metrics with static source structure
- Synthesize new metrics by combining metrics



HPCToolkit Workflow



- **Visualization**
  - — support top-down analysis with interactive viewer
  - — analyze results anytime, anywhere



Outline

- Brief overview of the HPCToolkit toolchain
- Three new components
  - — libmonitor
  - — call stack sampling
  - — hpcviewer
- New and notable
- Components

libmonitor: A Profiling Substrate

A substrate for profiling statically and dynamically linked code

- **Intercept and manage notable operations**
  - —monitor gains control & performs bookkeeping operations
  - —invokes a client callback function
  - —calls real version
- **Principal intercepts**
  - —library initialization, finalization
  - —process creation, finalization
    - – main, __libc_start_main, exit, _exit
    - – fork and exec
  - —threads
    - – initialize thread support
    - – parent: thread pre-create, thread-post-create
    - – child: thread creation, finalization
  - —dynamic library open, close
  - —mpi initialize, finalize

libmonitor: A Profiling Substrate (part 2)

- **Support functions**
  - —sigaction, signal
  - —system
  - —stack_bottom, process_bottom_frame
  - —mpi_comm_size, mpi_comm_rank
- **History: rewrite of Phil Mucci's libmonitor (UTK)**
  - — Mucci's libmonitor originally derived from HPCToolkit

Using libmonitor

- **Platforms**
  - — GNU/Linux (dynamic, static)
  - — BG/P, CNL, Catamount (static)
  - — any Unix (static)
- **Repository host: SciDAC Outreach Center**
  - — https://outreach.scidac.gov/projects/libmonitor
  - — svn repository
- **License: BSD (3-clause)**

libmonitor: Building and Running

- Autoconf and Automake
- Dynamic (libmonitor.so)
  - LD_PRELOAD override __libc_start_main()
  - works with unmodified, optimized binary
  - limited to GNU/Linux with __libc_start_main
- Static (libmonitor_wrap.a)
  - ld --wrap main
  - define __wrap_main(), refer to __real_main()
  - works on most any Unix
  - requires re-linking application

libmonitor Example

Use libmonitor as glue between application and profiler

```
void *
monitor_init_process(int argc, char **argv, void *data)
{
   initialize_profiling();
   start_profiling();
   return NULL;
}

void
monitor_fini_process(int how, void *data)
{
   stop_profiling();
   print_results();
 }
```

libmonitor Technical Points

- Lazy dlsym of library functions (e.g. pthread_create)
- Installs signal handler for every signal
  - offers signal to client first, then application
- Catches all types of exit, _exit, signals, exec, pthread_exit, pthread_cancel
- Keeps list of threads for thread shoot down
  - mechanism to get into thread at exit (via signal)
- Provides a thread-local pointer for each thread

libmonitor To-do List

- **Revisit some callback functions**
  - — more general access to library functions
  - — provide real (unmonitored) versions of overrides
- **Handle system, compiler quirks**
  - — when pthread_create() called before main().
  - — libc system() calls hidden fork()
  - — ia64 __libc_start_main() misbehaves
- **Better MPI support**

Outline

- Brief overview of the HPCToolkit toolchain
- Three new components
  - libmonitor
  - call stack sampling
  - hpcviewer
- New and notable
- Components

Measurement Challenges

Performance often depends upon context

- **Layered design**
  - —application frameworks, math libraries, communication libraries
- **Generic programming, e.g. C++ templates**
  - —both data structures and algorithms
- **Context-sensitive optimization**
  - —e.g. inlining
- **Goals**
  - —identify and quantify context-sensitive behavior
  - —differentiate between types of performance problems
    - – cheap procedure called many times
    - – expensive procedure called few times

Call Path Profiling

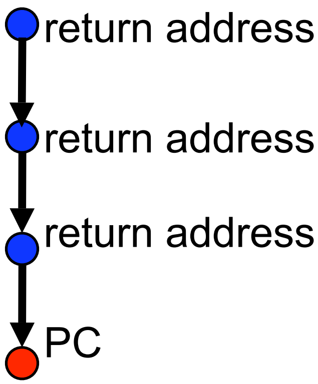
- **No instrumentation**
  - —statistical sampling of hardware performance counter overflows
  - —gather calling context information using stack unwinding
  - —overhead proportional to sampling frequency
    - – not calling frequency
- **Capture samples in full calling context**
  - —attribute sample to individual PC and source line
  - —associate costs with full calling context
    - – call sites too, not just callers

Novel Aspects of Our Approach

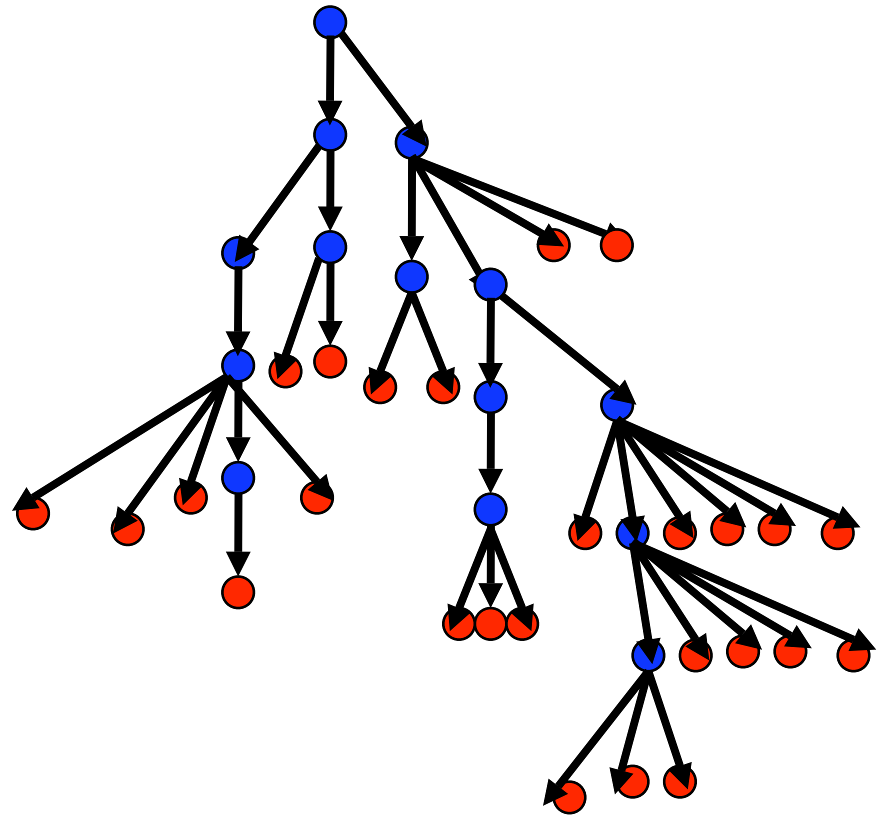
- **Unwind fully-optimized and even stripped code**
- **Cope with dynamically loaded shared libraries**
- **Integrate static & dynamic context information in presentation**
- **Differentiate between frequent and long calls**

A Call Path Profile

A call path sample



Calling Context Tree (CCT)



Unwinding Optimized Code

- **Optimized code presents challenges for unwinding**
  - —optimized code often lacks frame pointers
  - —no compiler information about epilogues
  - —routines may have multiple epilogues, multiple frame sizes
  - —code may be partially stripped: no info about function bounds
- **Difficulties**
  - —where is the return address of the current frame?
    - a register, relative to SP, relative to BP
  - —where is the FP for the caller's frame?
    - a register, relative to SP, relative to BP
- **Approach: use binary analysis to support unwinding**

Call Stack Unwinding of Optimized Code

```
cursor = initialize_cursor(machine_context)
do {
   ui = lookup_unwind_interval(PC)                       // splay tree
   if (ui is NULL)
      pb = lookup_procedure_bounds(PC)                   // binary search table
      if (pb is NULL)
         sg = lookup_segment_bounds(PC)                  // populate sg list
         if (sg is NULL) fail
         pb = compute_procedure_bounds(sg, PC)           // populate pb table
         if (pb is NULL) fail
         ui = compute_unwind_interval(pb, PC)            // populate ui splay tree
      if (ui is NULL) fail
   cursor = unwind_cursor(cursor, ui)                    // move cursor to caller
} while (more_frames_left(cursor))
```

Unwind Cursor

- **SP: stack pointer register for current frame**
- **BP: base pointer register for current frame**
- **IP: instruction pointer for the current frame**

Unwind Interval Information

- [start_addr, end_addr)
- ra_loc
  - —SP_RELATIVE: unwind using SP
    - – no BP initialized (yet); SP mods = only add/subtract constant
  - —STD_FRAME: unwind using SP or BP
    - – might set up BP as base pointer, but SP manipulation is transparent
  - —BP_FRAME: unwind using BP
    - – e.g. if BP overwritten in the routine
- bp_loc
  - —BP_UNCHANGED: BP on entry is still in BP
  - —BP_SAVED: BP on entry is saved in stack
  - —BP_OVERWRITTEN: BP is not useful for unwinding
- sp_ra_pos, sp_bp_pos: SP and BP offsets relative to SP
- bp_ra_pos, bp_bp_pos: SP and BP offsets relative to BP
- next, prev: pointers for doubly-linked list of intervals and splay tree edges

Unwinder Analyzer Details

Instructions tracked

- **call**: set high watermark interval for end of prologue
- **enter**: set up BP frame; adjust offsets accordingly
- **leave**: tear down BP frame
- **push**, pop: note SP change, check for BP save or restore
- **mov**: BP save/restore to memory; SP save/restore to/from BP
- **add**, **sub**: note if modify SP
- **conditional branch**: set high watermark interval for prologue
- **ret**: reset to canonical interval at next instruction
- **jmp**
  - —set high watermark interval for end of prologue
  - —reset to canonical interval at next instruction

Complications

- Invisible `alloca`
  - —PGI compiler uses support routines that move SP as side effect
  - —binary analysis may indicate STD_FRAME
  - —only unwind with BP will succeed
  - —approach needed:
    - – backtracking to use BP instead of SP when necessary
- Register-to-register moves of frame-relevant values
  - —mov %rbp,%rax
  - —mov %rax,0xb8(%rsp)
  - —must track register equivalences for frame relevant registers

SPEC 481.wrf, pathscale 3.1 compiler

ADVANCE_PPT.in.MODULE_PHYSICS_ADDTENDC

- Unconditional control transfers
  - — reset to “canonical interval” interval for following instruction

Finding Procedure Bounds

- Unwind interval analyzer requires function start and end
- Normally, obtain these from the symbol table
- If symbol table is partially stripped, need to recover them
- Approach
  - —seed process with dynamic symbols
  - —segment boundaries (PLT, INIT, FINI, TEXT)
  - —scan code segments (PLT, INIT, FINI, TEXT)
    - – build candidate set
      - note every instruction that is a target of a call
      - note every instruction that follows an unconditional control transfer and pad bytes
    - – build filter set
      - every instruction that is within the span of a conditional branch
  - —output filtered candidate set

Dynamically Loaded Code

- Issue: new code may be loaded/unloaded at any time
- When a new module is loaded
  - —indicate that a module is being loaded
  - —load the module (and any of its dependents)
  - —note new code segment mappings
  - —build table of new procedure bounds
- When a module is unloaded
  - —mark end of profiler epoch: code addresses no longer apply
  - —flush stale cached information

Call Stack Unwinding Effectiveness

- **Test cases using SPEC CPU 2006 benchmarks**
  - —combination of spec train and ref tests
  - —compiled with intel 10.0.23 compiler
  - —compiled with pathscale 3.1 compiler
  - —compiled with PGI 7.0.3 compiler
- **11M samples, dropped 234 samples**
  - —we know the issues and expect to reduce this further

Overhead on Opteron

benchmark	intel	intel-cs	intel cs % ovhd	path	path-cs	path cs %ovhd
400.perlbench	1037.64	1049.35	1.13	1078.79	1104.68	2.40
401.bzip2	1397.57	1318.45	-5.66	1328.97	1336.70	0.58
403.gcc	1320.43	1209.24	-8.42	1247.53	1208.62	-3.12
429.mcf	1687.41	1703.71	0.97	1084.54	1100.16	1.44
445.gobmk	907.38	929.10	2.39	1140.31	1159.96	1.72
456.hmmer	1098.40	1107.32	0.81	867.04	879.87	1.48
458.sjeng	1218.29	1228.91	0.87	1307.24	1314.02	0.52
462.libquantum	1850.02	1861.00	0.59	1738.72	1750.45	0.67
464.h264ref	1897.79	1909.53	0.62	1544.49	1596.84	3.39
471.omnetpp	990.72	1002.84	1.22	993.98	1003.14	0.92
473.astar	1069.19	1076.91	0.72	1026.62	1032.83	0.60
483.xalancbmk	906.09	1019.04	12.47	920.67	1063.77	15.54
999.specrand	0.23	1.79	685.99	0.22	1.47	559.62
410.bwaves	1277.41	1296.05	1.46	1169.10	1191.95	1.95
416.gamess	1769.74	1777.28	0.43	2099.06	2122.12	1.10
433.milc	1137.34	1113.90	-2.06	1071.40	1067.53	-0.36
434.zeusmp	1064.95	1096.93	3.00	1092.61	1094.73	0.19
435.gromacs	808.66	814.71	0.75	824.94	829.14	0.51
436.cactusADM	1276.82	1320.08	3.39	1229.49	1245.18	1.28
437.leslie3d	1217.68	1236.99	1.59	1172.38	1194.29	1.87
444.namd	922.21	925.75	0.38	951.64	953.07	0.15
447.dealII	1057.41	1037.34	-1.90	854.29	861.18	0.81
450.soplex	1083.53	1107.86	2.25	1169.66	1161.72	-0.68
453.povray	437.44	443.40	1.36	442.66	443.51	0.19
454.calculix	1484.73	1494.84	0.68	1574.34	1584.51	0.65
459.GemsFDTD	1883.90	1444.66	-23.32	1448.85	1404.32	-3.07
465.tonto	1031.49	1029.54	-0.19	1005.57	969.97	-3.54
470.lbm	1526.78	1507.01	-1.30	1463.73	1372.36	-6.24
481.wrf	1218.33	1027.28	-15.68	1088.75	1078.20	-0.97
482.sphinx3	2014.36	2058.90	2.21	1988.31	1842.07	-7.35
998.specrand	0.23	1.66	626.90	0.28	1.47	423.30
total	36594.19	36151.36	-1.21	34926.18	34969.85	0.13

SPEC CPU2006 Benchmarks
Opteron 246
200 samples/sec
ref runs

compilers
pathscale 3.1
intel 10.0.23

note
not yet memoizing path prefixes

Outline

- Brief overview of the HPCToolkit toolchain
- Three new components
  - libmonitor
  - call stack sampling
  - hpcviewer
- New and notable
- Components

hpcviewer

- Open Source
- Built on top of Eclipse platform
  - Independent application: Rich client platform
- Available on most platforms
  - x86 (32 and 64), PPC
  - Linux/GTK, Mac, Windows
  - Ongoing work: Itanium
- Requirements
  - Java 1.5
  - GTK for Linux

Derived Metrics

- **Allow users to define new metrics**
  - **Use a formula to compose existing metrics**
    - **floating point waste: (2 x Cycle) - FP_Ins**
    - **performance losses: min($1-$2, 0)**
- **Ongoing work**
  - **predefined derived metrics**
    - **performance losses, bandwith consumed, ...**
  - **storing derived metrics into a database**

Hot Call Paths

- Account for cost of performance hot-spots
- Show the chain of responsibility for costs
- How long is the chain?
  - — compare parent and child values
  - — if the difference is greater than a threshhold (50%)
    - – continue the path through that child

Demo

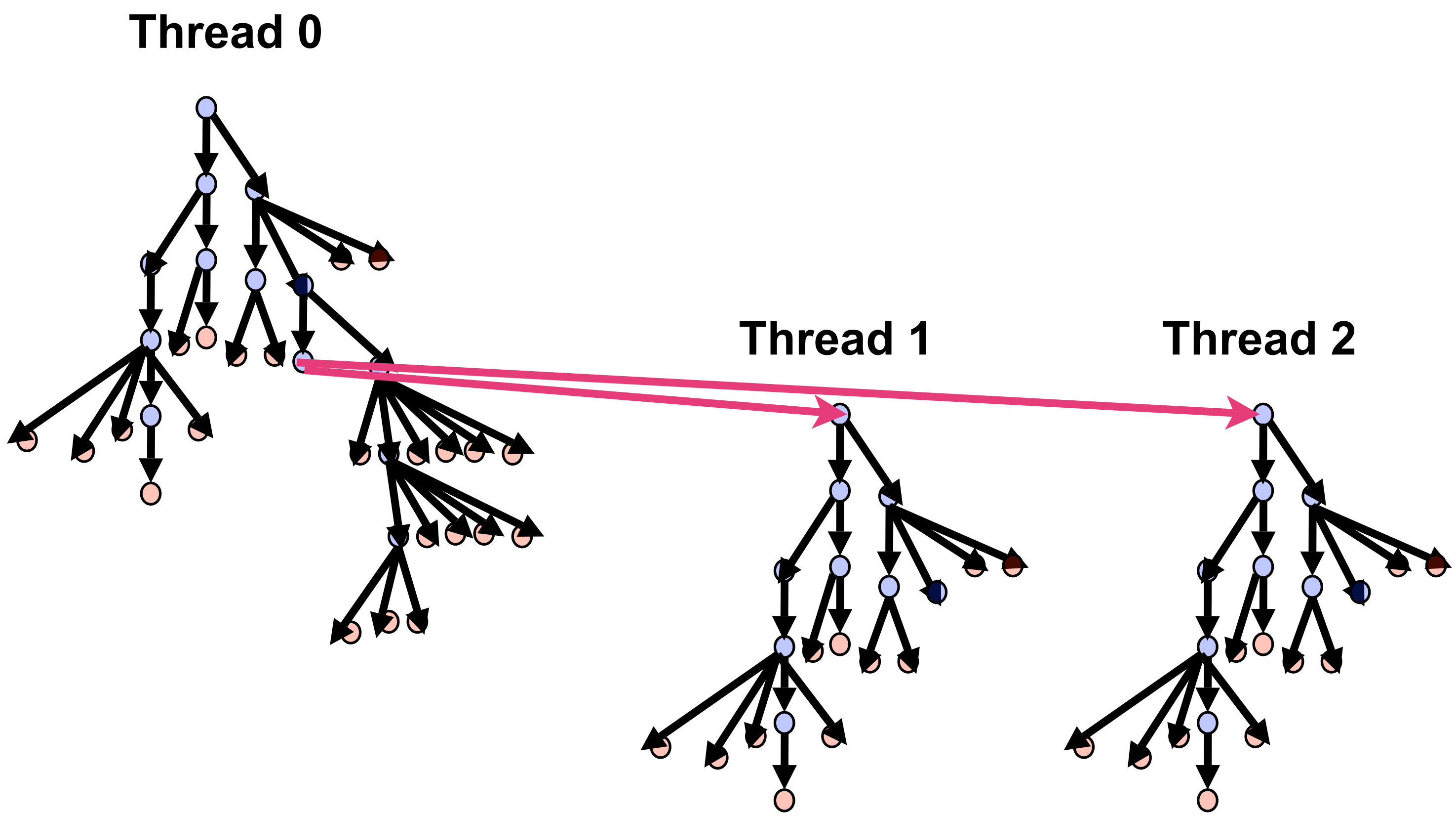
Outline

- Brief overview of the HPCToolkit toolchain
- Three new components
  - libmonitor
  - call stack sampling
  - hpcviewer
- New and notable
- Components

New and Notable

- **First-generation unwinder for BG/P**
  - — needs binary analysis to identify routine prologues for completeness
- **Pinpointing bottlenecks in multithreaded code**
  - — insufficient parallelism
  - — parallel overhead
- **Detailed modeling of performance bottlenecks**
  - — provide insight into why performance is bad

Integrated View of Multiple Threads



Outline

- Brief overview of the HPCToolkit toolchain
- Three new components
  - libmonitor
  - call stack sampling
  - hpcviewer
- New and notable
- Components

Components We Use

- Symtab API - function bounds recovery
- Xed2 - first-party binary analysis of x86, function bounds recovery
- binutils - binary analysis for structure recovery
- OpenAnalysis - CFG construction, interval analysis

Components We Want

- First-party binary analysis of PowerPC instructions
- Saving performance data from large-scale runs to disk
- Storing, indexing, and accessing performance data @ 100K
- Visualization components
  — integrate into Eclipse RCP