Software Support for Performance Measurement

Dr. John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu

Questions for Performance Measurement Tools

- **How much of resource X did my application consume?**
- **Where did my application consume resource X?**
- **Why did my application consume resource X where it did?**

Goals for Today

Understand

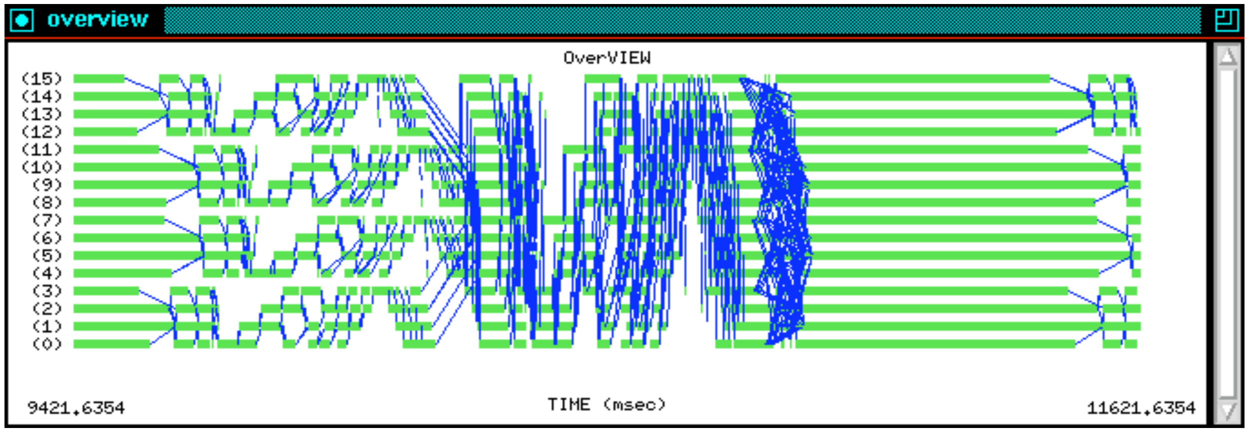
- **Performance monitoring software issues and approaches**
- **Performance instrumentation strategies**
- **Performance data collection strategies**
    - —Process-wide vs. system-wide profiling
    - —Flat (static) profiling
    - —Dynamic-context profiling

Measurement Issues

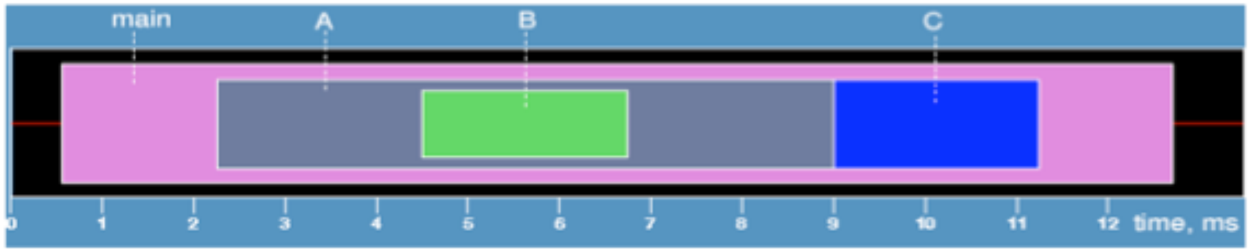
- Intrusion
  - —Heisenberg's uncertainty principle for software
- Diagnostic power
  - —how much detail is present in resource utilization info collected
  - —how much context is available to explain the resource utilization

Tracing

- **Detail about temporal ordering of events**
  - **useful for understanding process interactions via messages in parallel programs**



  - **useful for understanding temporal nature of subroutine calls**



- **High intrusion overhead if data not summarized on the fly**

Sample-based Profiling

Periodically interrupt application to record its PC

- **Advantages**
  - —low overhead
  - —no program instrumentation needed
  - —focuses attention on what is important
  - —pinpoints phenomena that are unexpectedly costly
- **Disadvantages**
  - —lacks information about temporal ordering

Profile Data Collection Strategies

- **Process-wide collection**
  - —advantage: no special privileges required
  - —disadvantage: myopic view of program behavior
    - – other activity could affect a program’s performance
- **System-wide collection**
  - —advantage: node-wide perspective on performance
  - —disadvantage: root privileges typically required to launch

Instrumentation Strategies

- Hand-inserted instrumentation: PAPI calipers
  —labor intensive. looks only where you do.
- Source-to-source: Oregon’s Tau
  —requires source. can disturb compiler optimization.
- Compiler-based instrumentation: gprof
- Static binary rewriting - DEC’s hiprof
  —many languages and programming models; threading a problem
- Load time initiation: SGI’s ssrun, Rice’s hpcrun, csprof
- On-the-fly binary modification: UMD/Wisconsin’s Dyninst
- Ubiquitous profiling without instrumentation: DCPI, oprofile

Dynamic Instrumentation

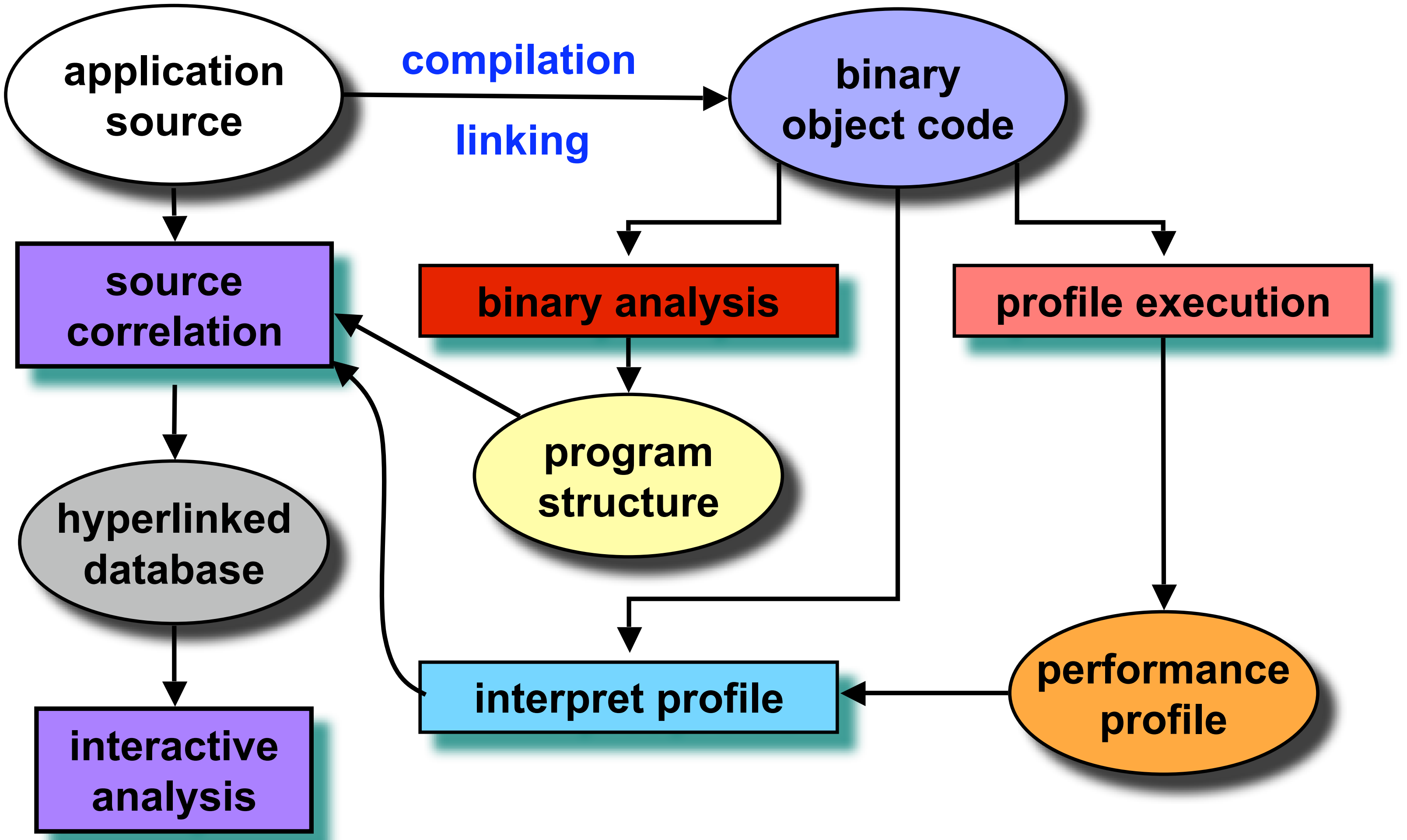
- On-the-fly in a running program
  - insert instrumentation
  - remove instrumentation
  - change instrumentation
- How?
  - patch a branch to an instrumentation code fragment into the executable whereever the instrumentation should be called
  - have the code fragment execute the overwritten instruction and any instrumentation
- Advantages of dynamic instrumentation
  - avoid dilating compile and/or link time
  - can profile without the source code (binary libraries)
  - no re-linking necessary
  - naturally handles dynamic libraries
  - instrumentation under program control can adapt
    - goal-driven monitoring and hypothesis testing
    - adaptive overhead reduction, e.g. code coverage tool

HPCToolkit: A full-featured flat profiler

HPCToolkit Goals

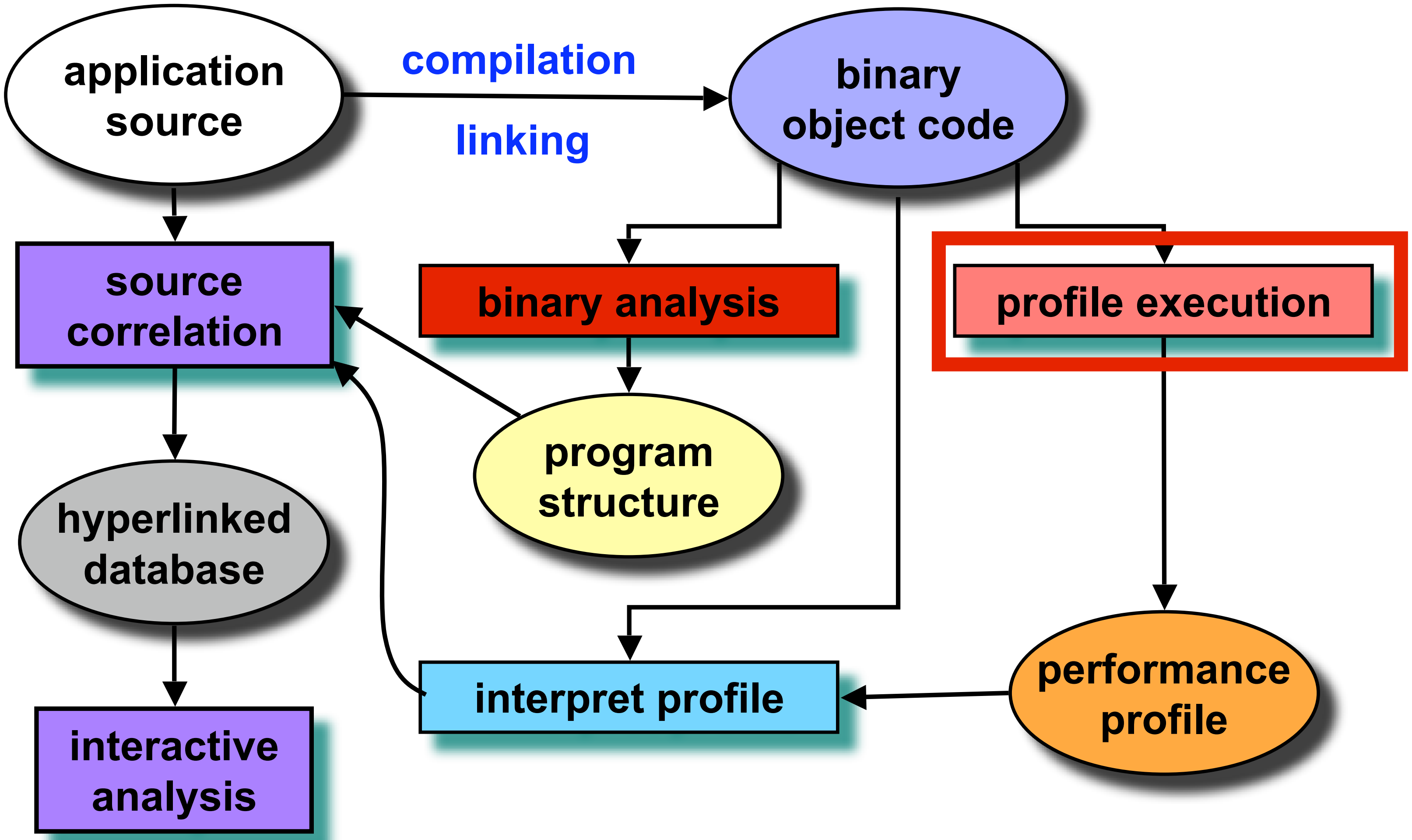
- Support large, multi-lingual applications
  - a mix of of Fortran, C, C++
  - external libraries (possibly binary only)
  - thousands of procedures, hundreds of thousands of lines
  - we must avoid
    - manual instrumentation
    - significantly altering the build process
    - frequent recompilation
- Collect execution measurements scalably and efficiently
  - don’t excessively dilate or perturb execution
  - avoid large trace files for long running codes
- Support measurement and analysis of serial and parallel codes
- Present analysis results effectively
  - top down analysis to cope with complex programs
  - intuitive enough for physicists and engineers to use
  - detailed enough to meet the needs of compiler writers
- Support a wide range of computer platforms

HPCToolkit System Workflow



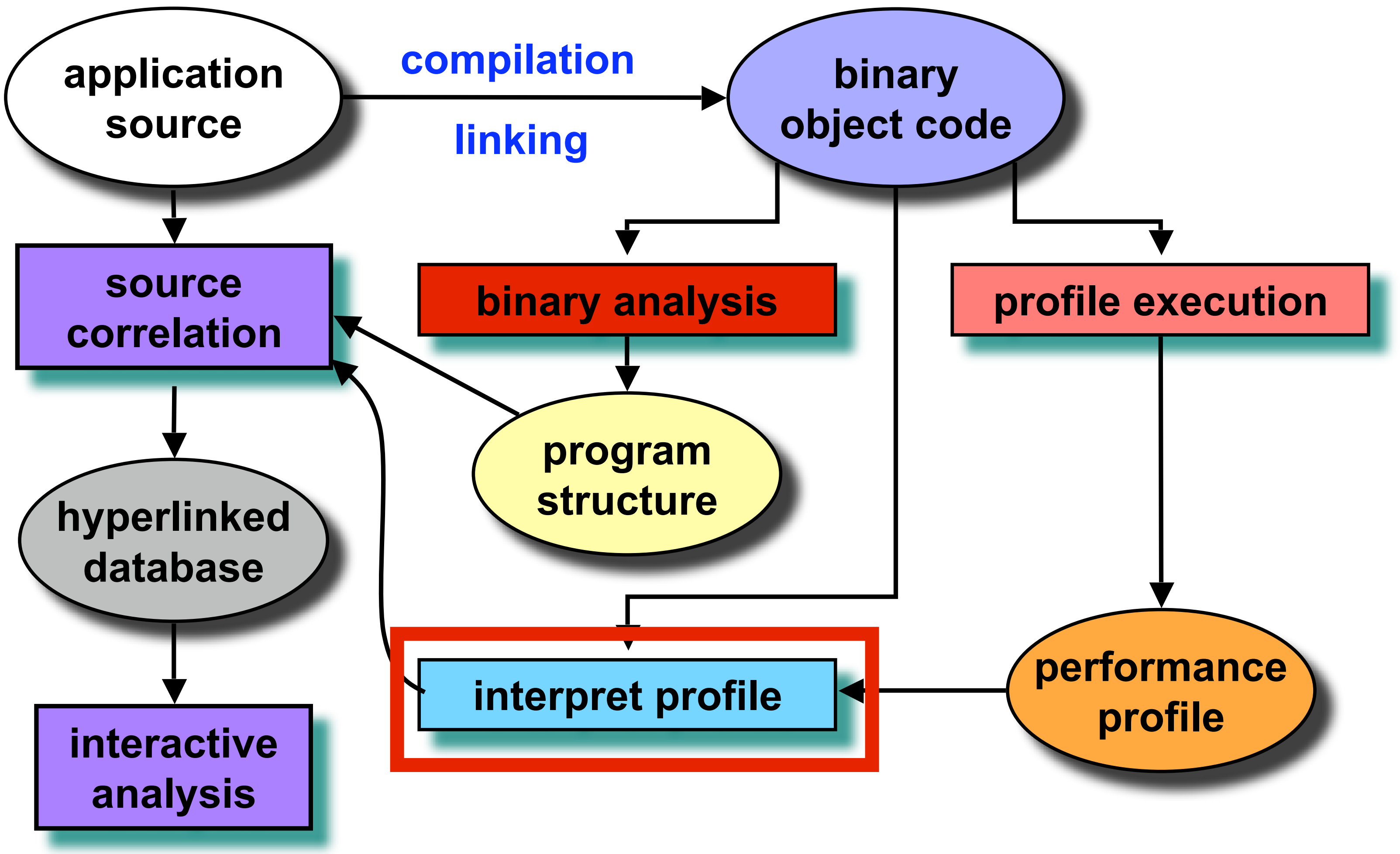
`http://www.hipersoft.rice.edu/hpctoolkit/`

HPCToolkit System Workflow



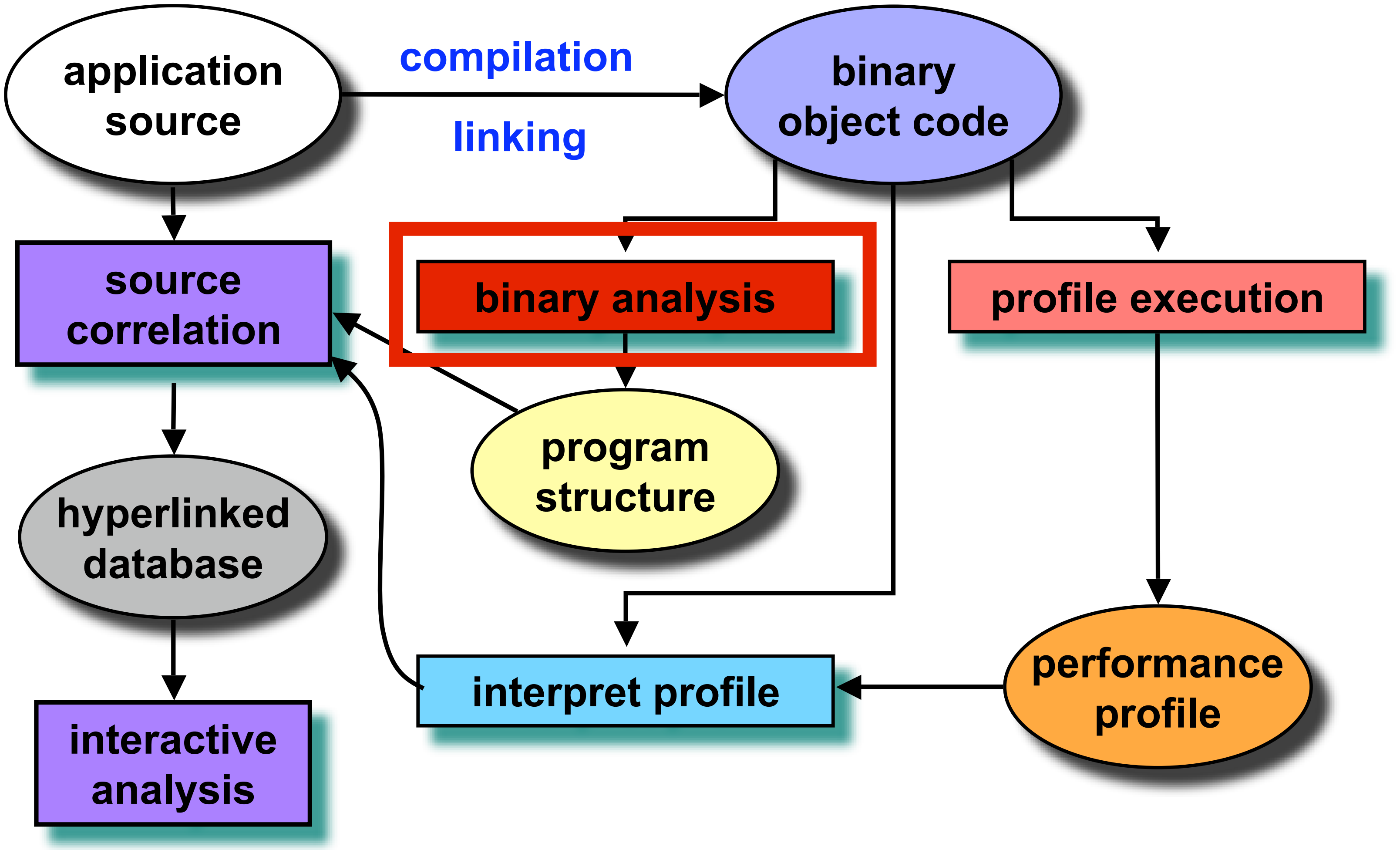
—launch unmodified, optimized application binaries
—collect statistical profiles of events of interest

HPCToolkit System Workflow



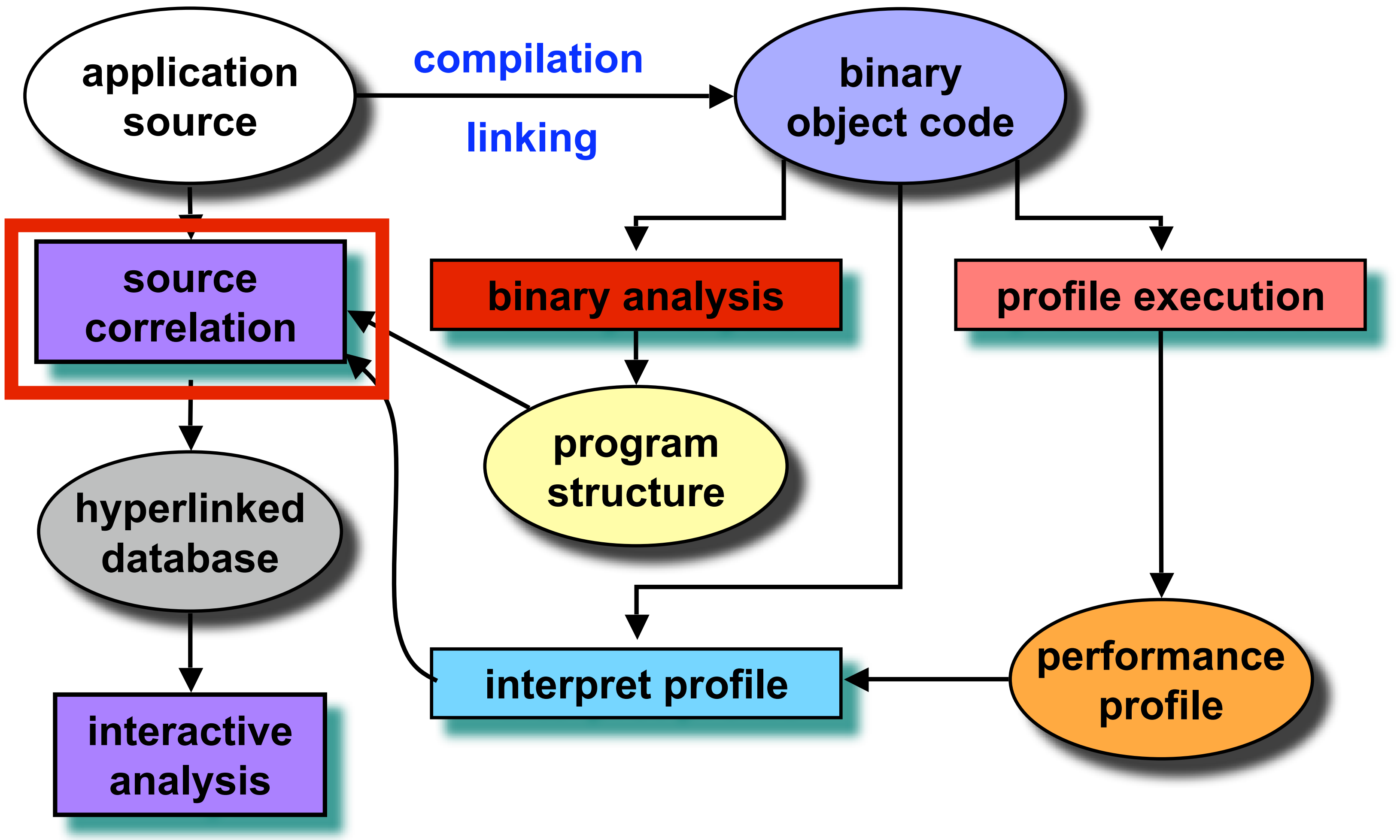
—decode instructions and combine with profile data

HPCToolkit System Workflow



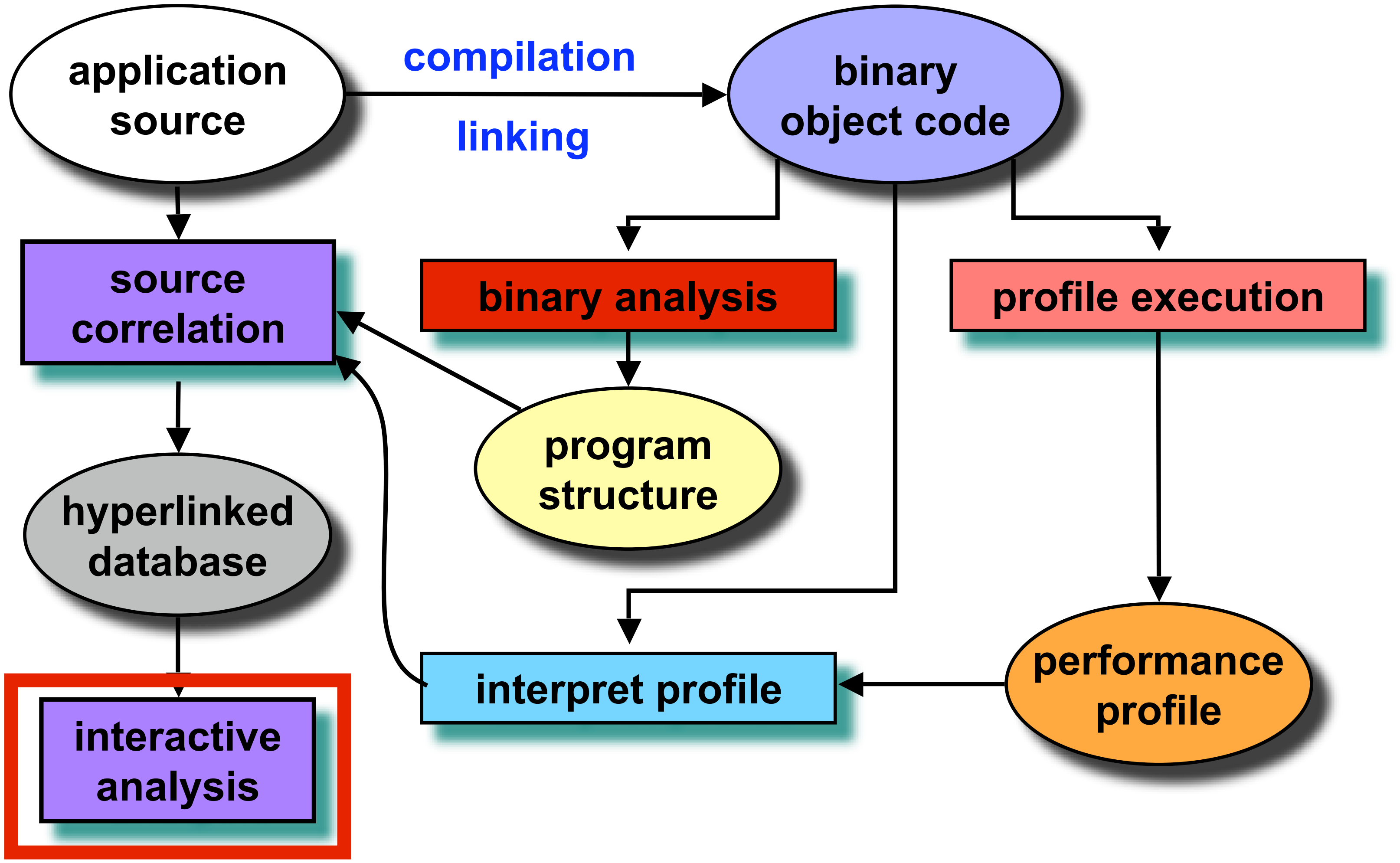
—extract loop nesting information from executables

HPCToolkit System Workflow



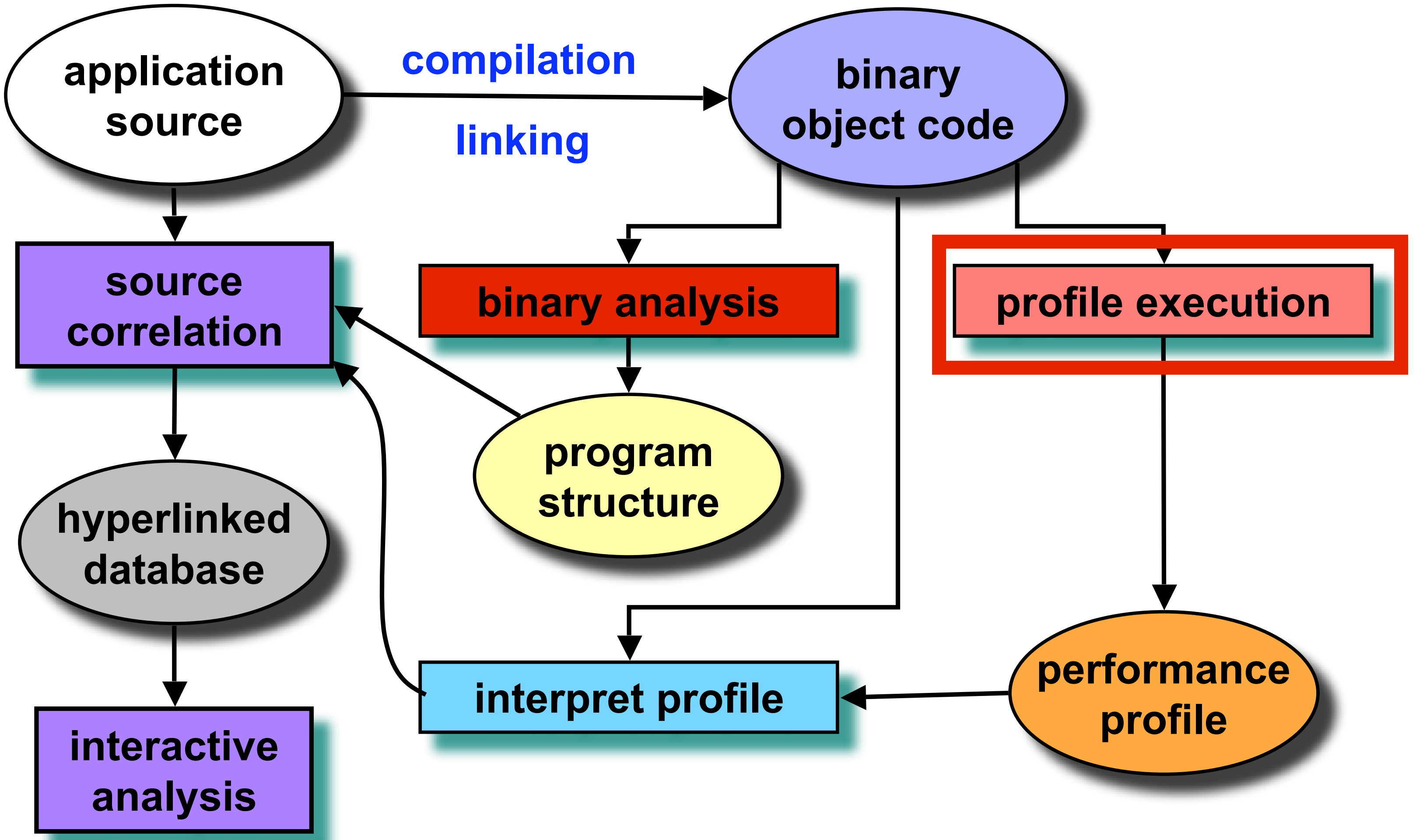
—synthesize new metrics by combining metrics
—relate metrics, structure, and program source

HPCToolkit System Workflow



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

HPCToolkit Workflow



hpcrun*

Capabilities

- Monitor unmodified, dynamically-linked application binaries
- Collect event-based program counter histograms
- Monitor multiple events during a single execution

Data collected for each (load module, event type) pair

- **Load module attributes**
  - name
  - start address
  - length
- **Event name**
- **Execution measurements**
  - A sparse histogram of PC values for which the event counter overflowed

*Only for use on Linux at present

`hpcrun` in Action

```
% ls
README
% hpcrun ls
ls.PAPI_TOT_CYC.master4.rtc.2866  README
```

Profile the default event (PAPI_TOT_CYC) for the `ls` command using the default period

Data file to which PC histogram is written for monitored event or events

Data file naming convention:
`command.eventname.nodename.pid`

A Running Example: sample.c

```
#define N (1 << 23)
#define T (10)
#include <string.h>
double a[N],b[N];
void cleara(double a[N]) {
  int i;
  for (i = 0; i < N; i++) {
    a[i] = 0;
  }
}
int main() {
  double s=0,s2=0; int i,j;
  for (j = 0; j < T; j++) {
    for (i = 0; i < N; i++) {
      b[i] = 0;
    }
    cleara(a);
    memset(a,0,sizeof(a));
    for (i = 0; i < N; i++) {
      s += a[i] * b[i];
      s2 += a[i] * a[i] + b[i] * b[i];
    }
  }
  printf("s %f s2 %f\n",s,s2);
}
```

- 2 user functions
- 2 calls to library functions
- Main has imperfectly nested loops with function calls

Using `hpcrun` to Collect a Performance Profile

```
% hpcrun -e PAPI_TOT_CYC:499997 -e PAPI_L1_LDM \
            -e PAPI_FP_INS -e PAPI_TOT_INS sample
```

Profile the “floating point instructions” using the default period

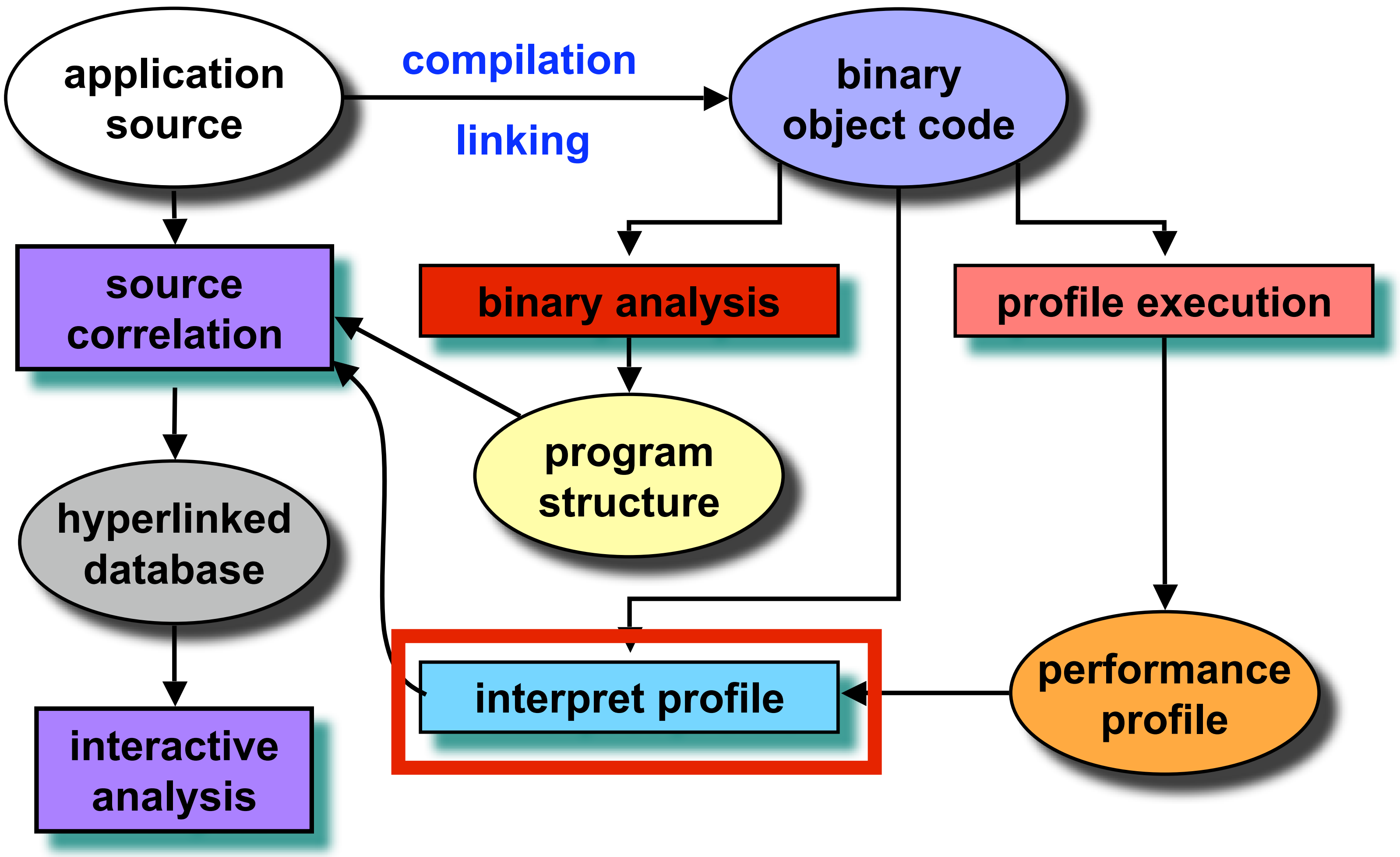
Profile the “total cycles” using period 499997

Profile the “total instructions” using the default period

Profile the “L1 data cache load misses” using the default period

Running this command on a machine “eddie” produced a data file
`sample.PAPI_TOT_CYC-etc.eddie.1736`

HPCToolkit Workflow



hpcprof*

Capabilities

- Read program counter histograms gathered by hpcrun
- Read debugging information in executable
- Use debugging information to map PC samples to
  - —Load module
  - —File
  - —Function
  - —Source line
- Output data in ASCII for human consumption
- Output data in XML for consumption by other tools

*Only for use on Linux at present

Using `hpcprof` to Interpret a Profile

```
% hpcprof -e sample sample.PAPI_TOT_CYC-etc.eddie.1736
Columns correspond to the following events [event:period]
  PAPI_TOT_CYC:32767 - Total cycles (264211 samples)
  PAPI_TOT_INS:32767 - Instructions completed (61551 samples)
  PAPI_FP_INS:32767 - Float point instructions (15355 samples)
  PAPI_L1_LDM:32767 - Level 1 load misses (6600 samples)

Load Module Summary:
  85.5% 100.0% 100.0%  99.9% /tmp/sample
  14.5%   0.0%   0.0%   0.1% /lib/libc-2.3.3.so

File Summary:
  85.5% 100.0% 100.0%  99.9% <</tmp/sample>>/tmp/sample.c
  14.5%   0.0%   0.0%   0.1% <</lib/libc-2.3.3.so>><unknown>
```

Continued …

`hpcprof`: Function and Line Output

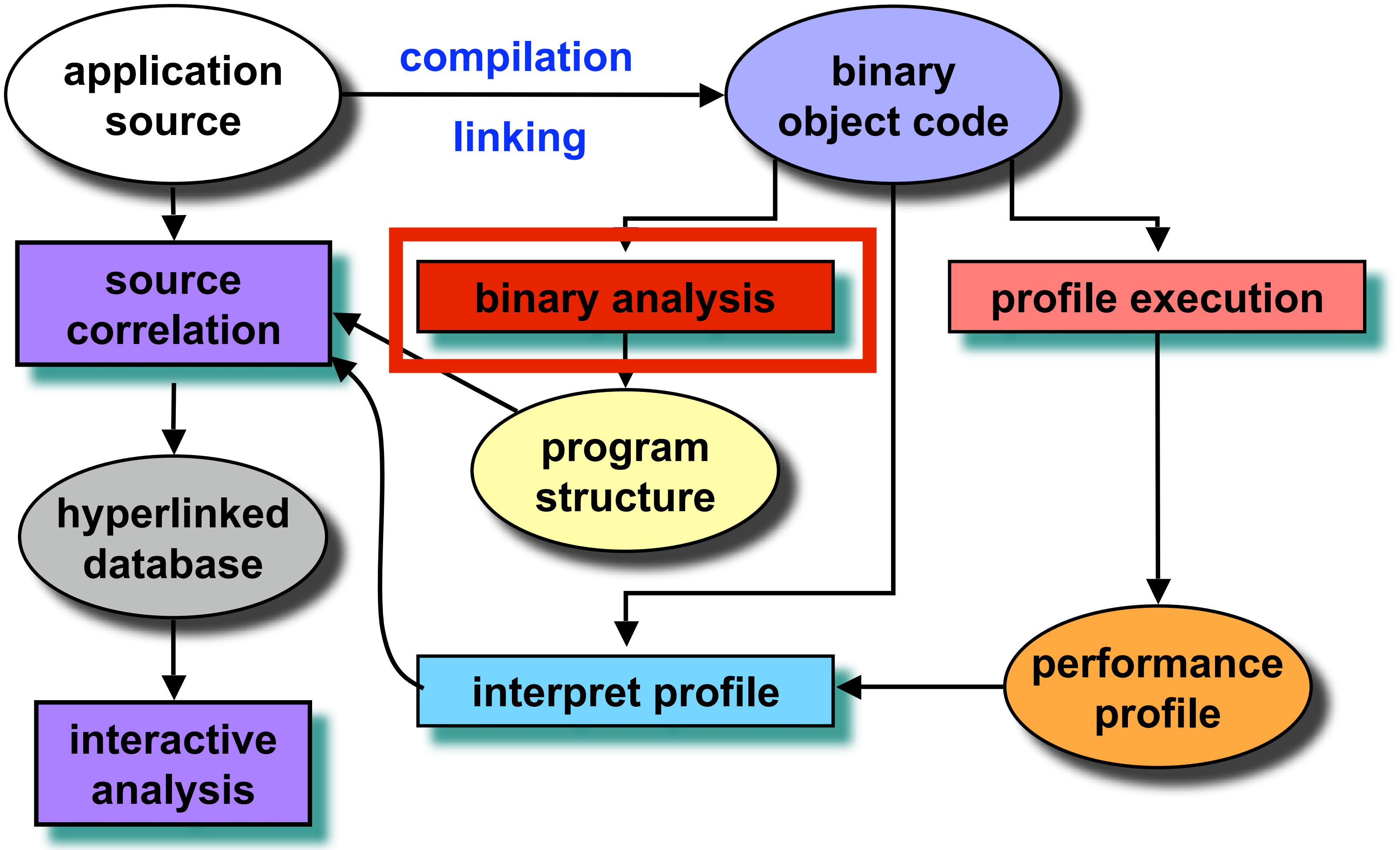
```
Function Summary:
  70.8%  83.3% 100.0%  99.7% <</tmp/sample>>main
  14.7%  16.7%   0.0%   0.2% <</tmp/sample>>cleara
  14.5%   0.0%   0.0%   0.1% <</lib/libc-2.3.3.so>>memset

Line Summary:
  37.2%  35.0%  55.9%  51.4% <</tmp/sample>>/tmp/sample.c:20
  13.4%  18.3%  30.2%  34.0% <</tmp/sample>>/tmp/sample.c:21
   5.6%   9.3%  13.9%  14.2% <</tmp/sample>>/tmp/sample.c:19
  11.6%  17.4%   0.0%   0.2% <</tmp/sample>>/tmp/sample.c:15
   8.8%  10.5%   0.0%   0.1% <</tmp/sample>>/tmp/sample.c:8
  14.5%   0.0%   0.0%   0.1% <</lib/libc-2.3.3.so>><unknown>:0
   5.9%   6.1%   0.0%   0.0% <</tmp/sample>>/tmp/sample.c:7
   3.1%   3.4%   0.0%   0.0% <</tmp/sample>>/tmp/sample.c:14
```

Continued …

hpcprof: Annotated Source File Output

```
File <</tmp/sample>>/tmp/sample.c with profile annotations.
     1                                 #define N (1 << 23)
     2                                 #define T (10)
     3                                 #include <string.h>
     4                                 double a[N],b[N];
     5                                 void cleara(double a[N]) {
     6                                   int i;
     7  14.5%  16.7%   0.0%   0.2%      for (i = 0; i < N; i++) {
     8                                     a[i] = 0;
     9                                   }
    10                                 }
    11                                 int main() {
    12                                   double s=0,s2=0; int i,j;
    13                                   for (j = 0; j < T; j++) {
    14   9.0%  14.6%   0.0%   0.1%        for (i = 0; i < N; i++) {
    15   5.6%   6.2%   0.0%   0.1%          b[i] = 0;
    16                                     }
    17                                     cleara(a);
    18                                     memset(a,0,sizeof(a));
    19   3.6%   5.8%  10.6%   9.2%        for (i = 0; i < N; i++) {
    20  36.2%  32.9%  53.2%  49.1%          s += a[i]*b[i];
    21  16.7%  23.8%  36.2%  41.2%          s2 += a[i]*a[i]+b[i]*b[i];
    22                                     }
    23                                   }
    24                                   printf("s %d s2 %d\n",s,s2);
    25                                 }
```

HPCToolkit Workflow



Why Binary Analysis?

Issues

- Line-level statistics offer a myopic view of performance
  - —Example: loads on one line, arithmetic on another.
- Event detection "skid" makes line-level metrics inaccurate
- Optimizing compilers rearrange and interleave code.
- Source-code analysis is language-dependent, compiler- (and compiler-version-) dependent.
- For scientific programs, loops are the unit of optimization and are where the action is

Approach

- Recover loop information from application binaries

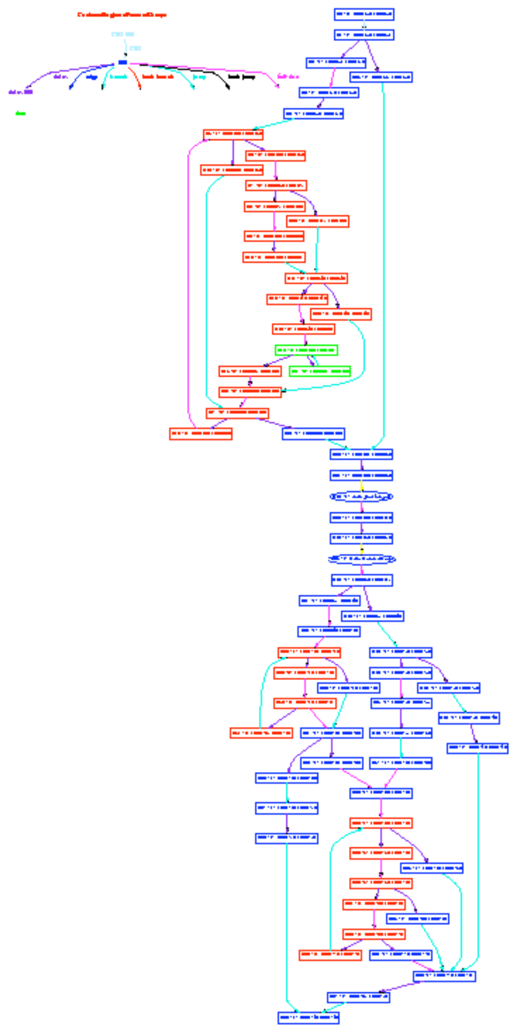
bloop

Extract loop structure from binary modules (executables, dynamic libraries)

Approach

- **Parse instructions in an application binary**
- **Identify branches and aggregate instructions into basic blocks**
- **Construct control flow graph using branch target analysis**
- **Use interval analysis to identify natural loop nests**
- **Map instructions to source lines with symbol table information**
  - **—Quality of results depends on quality of symbol table information**
- **Normalize results to recover source-level view of optimized code**
- **Elide information about source lines NOT in loops**
- **Output program structure in XML for consumption by hpcview**
- **Platforms: Alpha+Tru64, MIPS+IRIX, Linux+IA64, Linux+IA32, Solaris+SPARC**

Sample Flow Graph from an Executable



Loop nesting structure

- —blue: outermost level
- —red: loop level 1
- —green loop level 2

Observation:
optimization complicates program structure!

Normalizing Program Structure

Constraint: each source line may appear at most once

Coalesce multiple instances of lines

Repeat

- **(1) If the same line appears in different loops**
  - **find least common ancestor in scope tree; merge corresponding loops along the paths to each duplicate**
    - **purpose: re-roll loops that have been split**
- **(2) If the same line appears at multiple loop levels**
  - **discard all but the innermost instance**
    - **purpose: account for loop-invariant code motion**

Until a fixed point is reached

bloop in Action

```
% bloop sample > sample.bloop
<LM n="sample" version=”4.0">
   <F n="/tmp/sample.c">
      <P n="cleara" b="5" e="10">
         <L b="7" e="8">
            <S b="7" e="7"/>
            <S b="8" e="8"/> </L> </P>
      <P n="main" b="11" e="25">
         <L b="13" e="21">
            <S b="13" e="13"/>
            <L b="14" e="15">
               <S b="14" e="14"/>
               <S b="15" e="15"/> </L>
            <S b="17" e="17"/>
            <S b="18" e="18"/>
            <L b="19" e="21">
               <S b="19" e="19"/>
               <S b="20" e="20"/>
               <S b="21" e="21"/> </L> </L> </P> </F> </LM>
```

Load Module

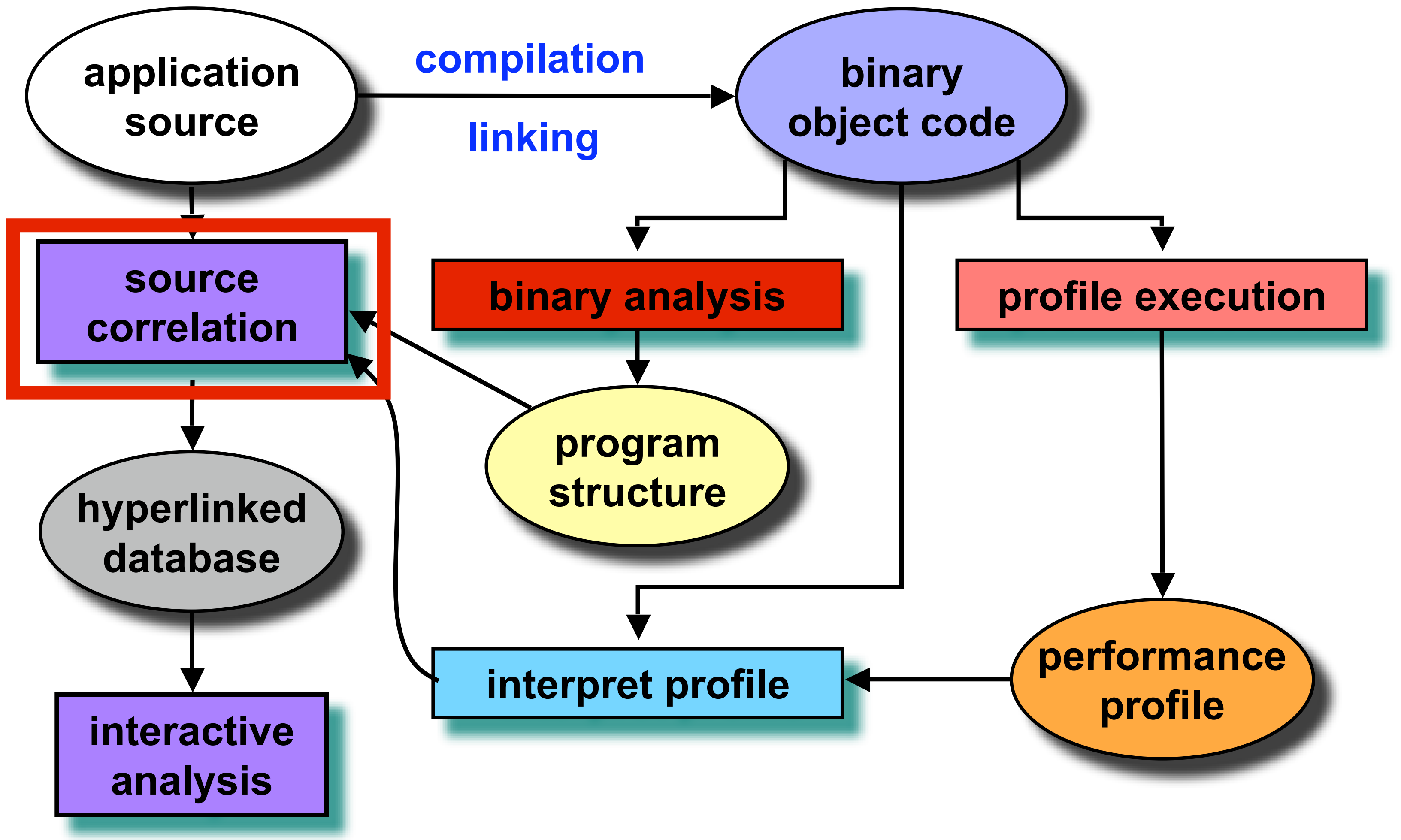
File

Procedure

Loop

Statement

HPCToolkit Workflow



Data Correlation

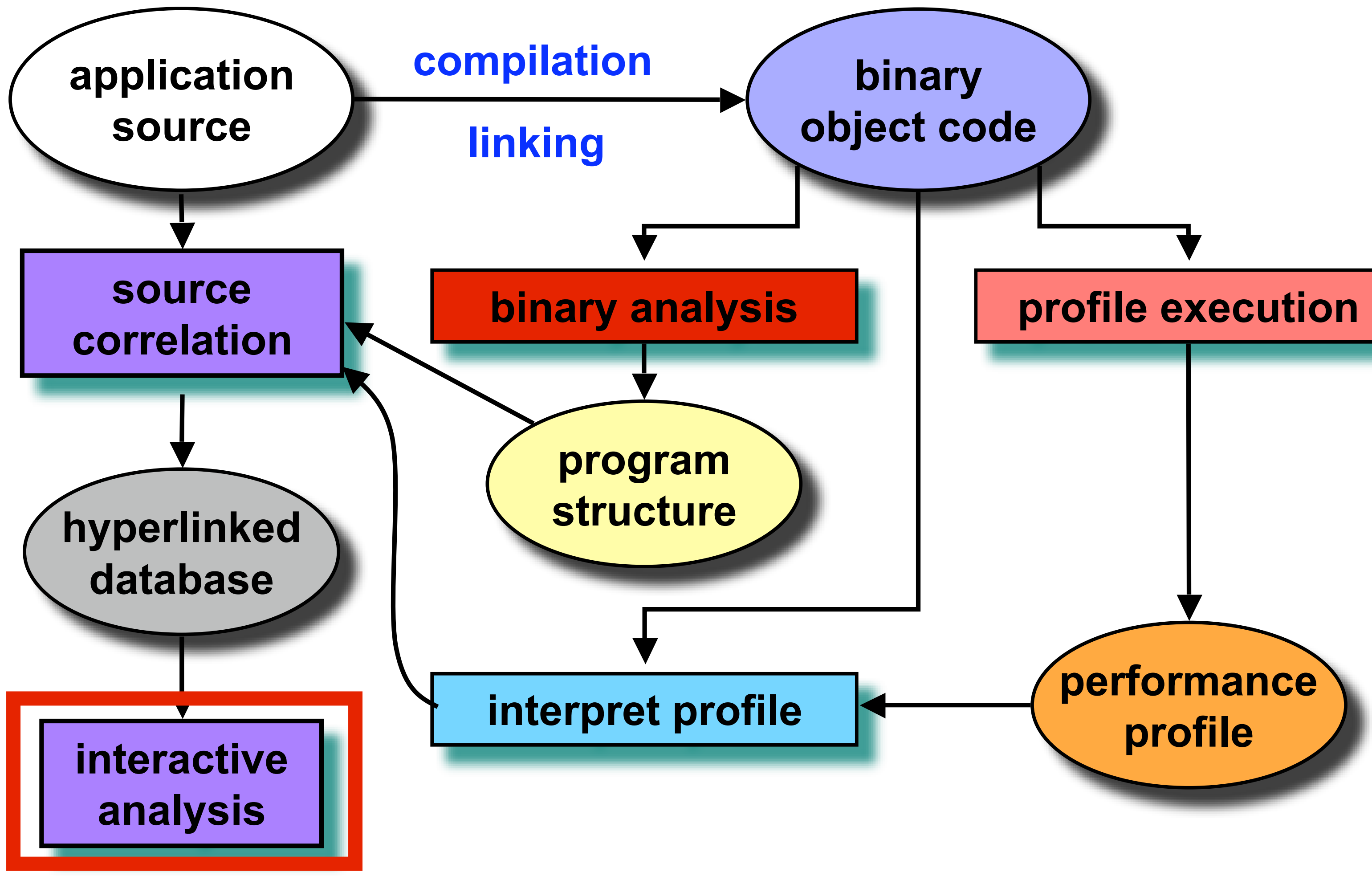
Motivation

- **Need to analyze data from multiple experiments together**
  - **Different architectures**
  - **Different input data sets**
  - **Different metrics: any one metric provides a myopic view**
    - **some measure potential *causes* (e.g. cache misses)**
    - **some measure *effects* (e.g. cycles)**
    - **e.g., cache misses are only a problem if latency not hidden**
- **Need to analyze metrics in the context of the source code**
- **Need to synthesize derived metrics, e.g. cache miss rate**
  - **Synthetic metrics are needed information**
  - **Automation eliminates mental arithmetic**
  - **Serve as a key for sorting**

`hpcview`

- `hpcview` correlates
  - —Metrics expressed as PROFILE XML documents
    - – Produced on Linux with `hpcprof`
  - —Program structure expressed as PROFILE XML documents
    - – Produced by `bloop`
  - —Program source code in any language
- `hpcview` produces a database containing both performance data and source code for exploration with `hpcviewer`
  - —Aggregates metrics up the program structure tree

HPCToolkit System Overview



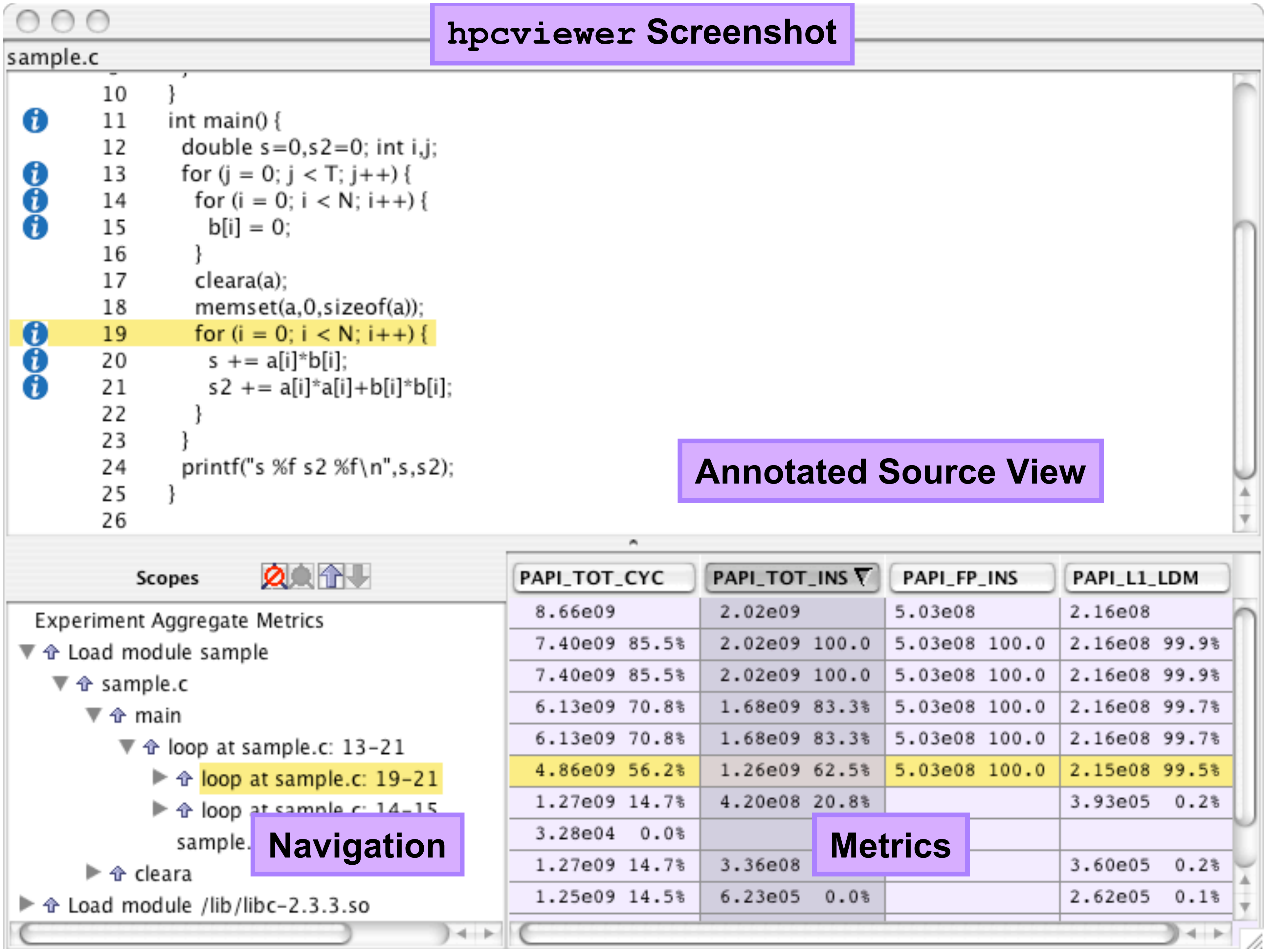
hpcviewer

Tool for interactive top-down analysis of performance data

- Execution metric data plus program source code
  - —Multiple metrics from one or more experiments
- Exploration is guided by rank-ordered table of metrics
  - —Can sort on any metric in ascending or descending order
- Hierarchical view of execution metrics
  - —Load modules
    - – Executable, shared libraries, operating system, other programs
  - —Files
  - —Procedures
  - —Loops
  - —Source lines

`hpcviewer` is written in Java and runs anywhere: your cluster, your desktop, your laptop

hpcviewer Screenshot



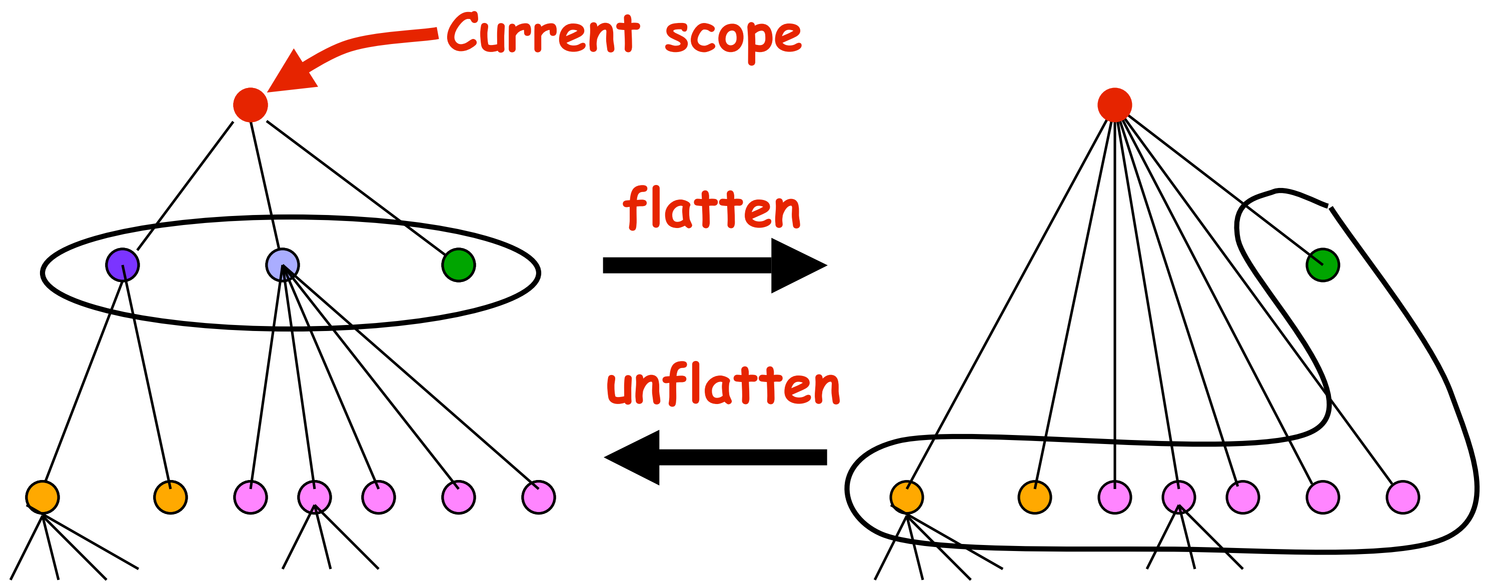
sample.c

```
10    }
11    int main() {
12      double s=0,s2=0; int i,j;
13      for (j = 0; j < T; j++) {
14        for (i = 0; i < N; i++) {
15          b[i] = 0;
16        }
17        cleara(a);
18        memset(a,0,sizeof(a));
19        for (i = 0; i < N; i++) {
20          s += a[i]*b[i];
21          s2 += a[i]*a[i]+b[i]*b[i];
22        }
23      }
24      printf("s %f s2 %f\n",s,s2);
25    }
26
```

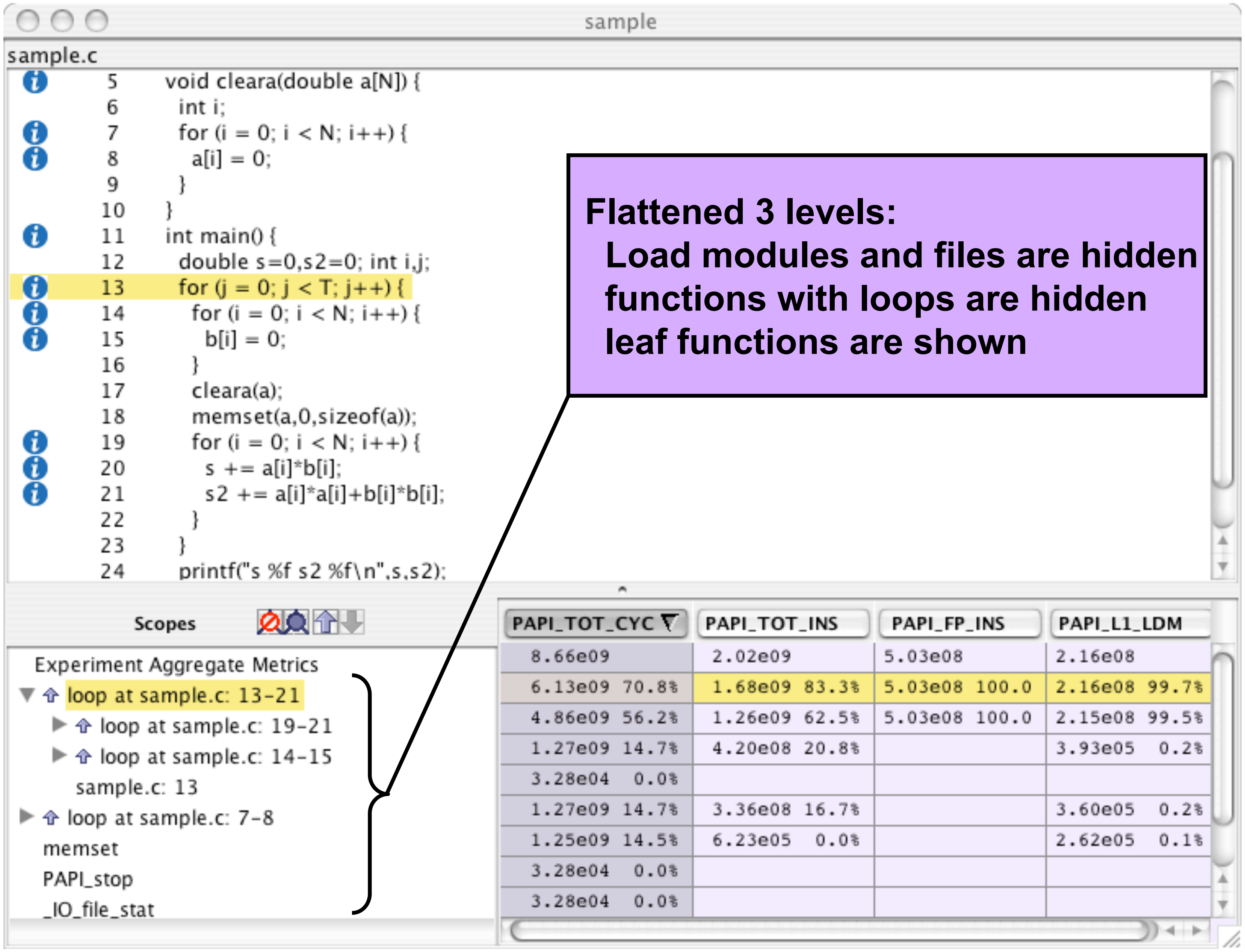
Scopes	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_FP_INS	PAPI_L1_LDM
Experiment Aggregate Metrics	8.66e09	2.02e09	5.03e08	2.16e08
Load module sample	7.40e09 85.5%	2.02e09 100.0	5.03e08 100.0	2.16e08 99.9%
sample.c	7.40e09 85.5%	2.02e09 100.0	5.03e08 100.0	2.16e08 99.9%
main	6.13e09 70.8%	1.68e09 83.3%	5.03e08 100.0	2.16e08 99.7%
loop at sample.c: 13-21	6.13e09 70.8%	1.68e09 83.3%	5.03e08 100.0	2.16e08 99.7%
loop at sample.c: 19-21	4.86e09 56.2%	1.26e09 62.5%	5.03e08 100.0	2.15e08 99.5%
loop at sample.c: 14-15	1.27e09 14.7%	4.20e08 20.8%		3.93e05 0.2%
sample.	3.28e04 0.0%			
cleara	1.27e09 14.7%	3.36e08		3.60e05 0.2%
Load module /lib/libc-2.3.3.so	1.25e09 14.5%	6.23e05 0.0%		2.62e05 0.1%

Flattening for Top Down Analysis

- **Problem**
  - —strict hierarchical view of a program is too rigid
  - —want to compare program components at the same level as peers
- **Solution**
  - —enable a scope's descendants to be flattened to compare their children as peers



Flattening in Action



```
 5   void cleara(double a[N]) {
 6     int i;
 7     for (i = 0; i < N; i++) {
 8       a[i] = 0;
 9     }
10   }
11   int main() {
12     double s=0,s2=0; int i,j;
13     for (j = 0; j < T; j++) {
14       for (i = 0; i < N; i++) {
15         b[i] = 0;
16       }
17       cleara(a);
18       memset(a,0,sizeof(a));
19       for (i = 0; i < N; i++) {
20         s += a[i]*b[i];
21         s2 += a[i]*a[i]+b[i]*b[i];
22       }
23     }
24     printf("s %f s2 %f\n",s,s2);
```

Scopes	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_FP_INS	PAPI_L1_LDM
Experiment Aggregate Metrics	8.66e09	2.02e09	5.03e08	2.16e08
loop at sample.c: 13-21	6.13e09 70.8%	1.68e09 83.3%	5.03e08 100.0	2.16e08 99.7%
loop at sample.c: 19-21	4.86e09 56.2%	1.26e09 62.5%	5.03e08 100.0	2.15e08 99.5%
loop at sample.c: 14-15	1.27e09 14.7%	4.20e08 20.8%		3.93e05 0.2%
sample.c: 13	3.28e04 0.0%			
loop at sample.c: 7-8	1.27e09 14.7%	3.36e08 16.7%		3.60e05 0.2%
memset	1.25e09 14.5%	6.23e05 0.0%		2.62e05 0.1%
PAPI_stop	3.28e04 0.0%			
_IO_file_stat	3.28e04 0.0%			

Flattened 3 levels:

Load modules and files are hidden

functions with loops are hidden

leaf functions are shown

Summary of HPCToolkit Functionality

- Top down analysis focuses attention where it belongs
  - —sorted views put the important things first
- Integrated browsing interface facilitates exploration
  - —rich network of connections makes navigation simple
- Hierarchical, loop-level reporting facilitates analysis
  - —more sensible view when statement-level data is imprecise
- Binary analysis handles multi-lingual applications and libraries
  - —succeeds where language and compiler based tools can't
- Sample-based profiling, aggregation and derived metrics
  - —reduce manual effort in analysis and tuning cycle
- Multiple metrics provide a better picture of performance
- Multi-platform data collection
- Platform independent analysis tool

Typical Uses for HPCToolkit

- Identifying unproductive work
  - —where is the program spending its time not performing FLOPS
- Memory hierarchy issues
  - —bandwidth utilization: misses x line size/cycles
  - —exposed latency: ideal vs. measured
- Cross architecture or compiler comparisons
  - —what program features cause performance differences?
- Gap between peak and observed performance
  - —loop balance vs. machine balance?
- Evaluating load balance in a parallelized code
  - —how do profiles for different processes compare

HPCToolkit Futures

- **What: better attribution of costs to code with multiple instances**
  - —inlined functions
  - —statement functions
  - —templates
  - —software pipelined loops (prologue, steady state, epilogue)
- **How: combine binary analyzer with profiler to identify instances of replicated code and account for them separately**

Profiling with Dynamic Context

Challenges for Analysis of Modern Programs

- Common features of modern programs
  - Layered design
  - Modular structure of reusable components
    - object-oriented components
    - data structure templates
      - e.g. containers
    - algorithm templates
  - Recursive functions
- Understanding performance requires
  - understanding interplay of layers
  - understanding context-dependent behavior of components
    - e.g. list templates used for different types of lists
  - understanding structure of recursive solution
    - quicksort: balanced or imbalanced splits

calling context required for detailed understanding

gprof

Ubiquitous call graph profiler designed in 1982

- **Counts of routine invocations**
- **Timing information for statements or routine invocations**
  - **—static structure presentation**
    - **associate resource consumption (time) with statements or routine invocations.**
  - **—dynamic structure presentation**
    - **resource consumption by self**
    - **resource consumption by called procedures**

Types of Call Graphs

- Complete call graph
  - —all routines and potential arcs
  - —could be determined by dataflow analysis
- Static call graph
  - —misses arcs representing calls to function variables
  - —readily determinable from static program text
- Dynamic call graph ← `gprof` builds these
  - —only includes nodes and arcs observed during execution
  - —may include arcs representing calls to function variables
  - —determined only via profiling

gprof: Under the Hood

- **Program instrumentation**
  - compiler augments program with call to `mcount` in function prologue
  - advantages of monitoring routine
    - late binding of program instrumentation (link time)
- **Run-time data structure**
  - two level hash table
    - primary table: call site address -> secondary table
    - secondary table: callee address -> count of invocations
- **Data collection: two parts**
  - collect PC histogram using statistical sampling driven by SIGPROF delivery from interval timer
  - in each call to `mcount` from function `f`
    - (conceptually) increment count of entries to `f`
    - increment count of invocations of `f associated with call graph edge (identified by f's return address)`

gprof Execution Time Monitoring

- **Initiation**
  - —Initialize monitoring data structures
  - —Initiate sampling
- **During execution**
  - —each function invokes monitoring routine from its function prologue
  - —samples accumulate in PC histogram
- **Finalization**
  - —organize non-zero PC histogram data into a sparse representation
  - —write sparse PC histogram and edge counts to output file

Why Exact Call Counts?

- Use them for debugging
  - —is a function unexpectedly called?
  - —is test coverage exhaustive?

gprof limitations

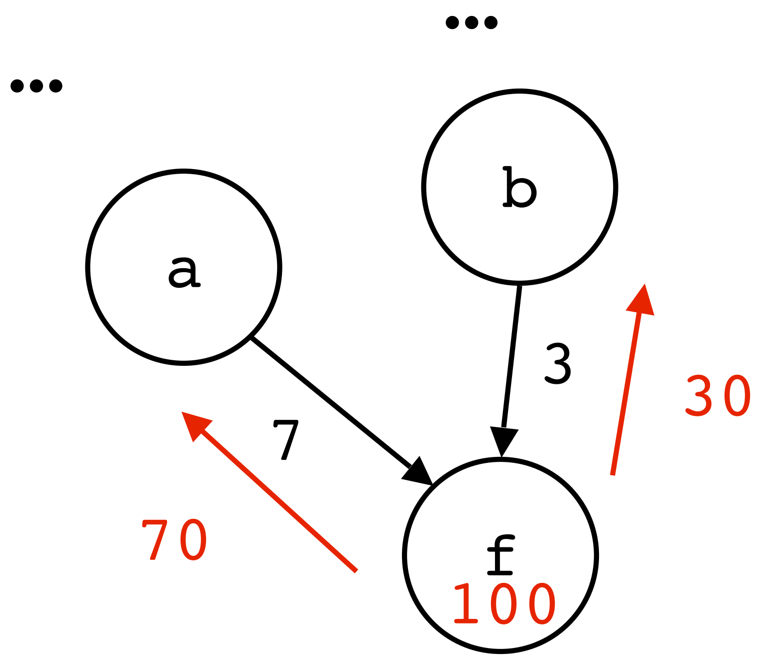
- **Unless prologue code is compiled in, edges are not counted**
- **Not all callers can be identified ⇒ spontaneous invocations**
  - **e.g. exception handlers have non-standard calling conventions; caller unknown**

Points to Ponder

- Execution counts not necessarily proportional to time
- Each call may take different amount of time
- For layered abstractions, time spent in an abstraction may not be adequately reflected

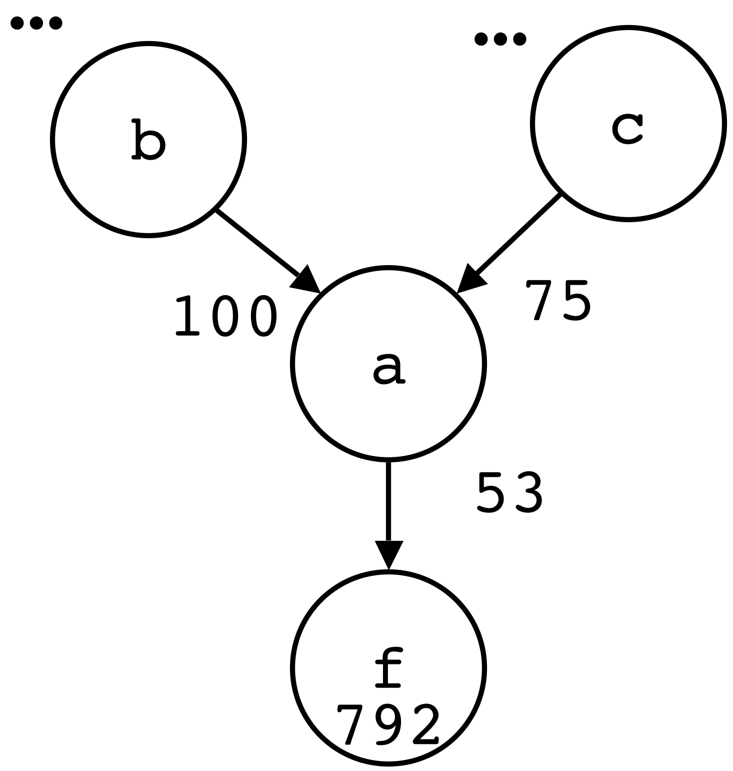
gprof Key Assumption

- If f was invoked K times for a total execution time T
  - —assume each invocation of f took T/K time
- Attribute fraction of cost associated with f to each caller according to call frequency



Shortcomings of Call Graphs

- Consider the following call graph

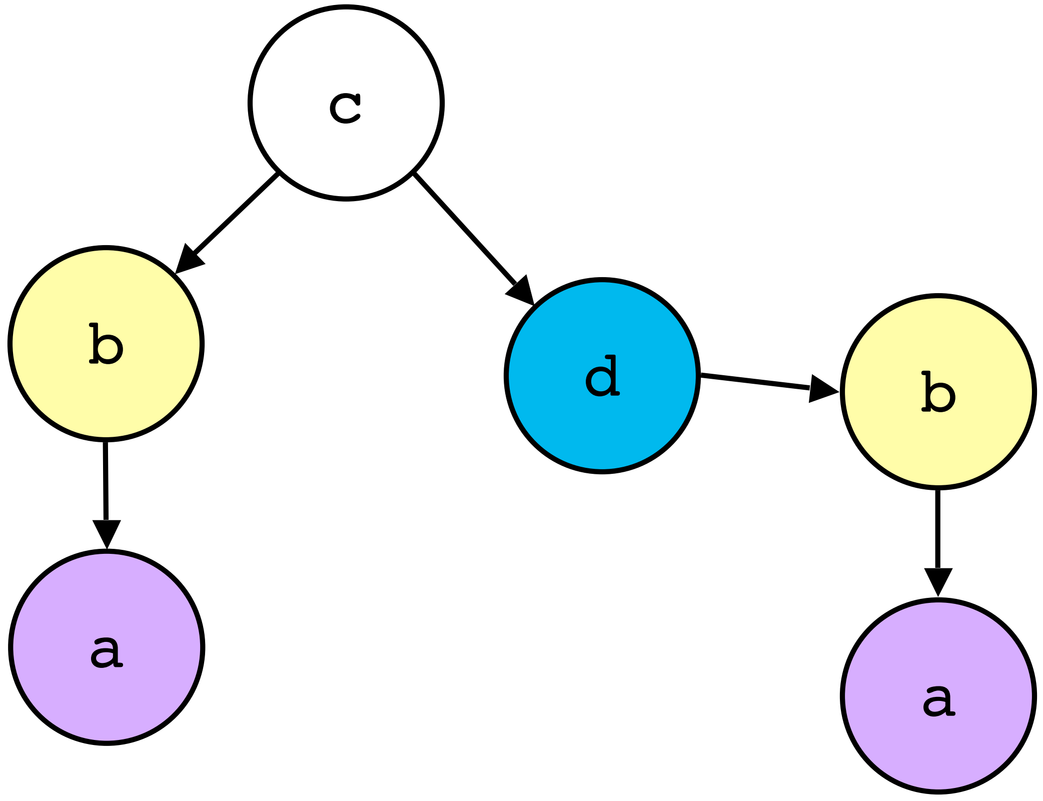


- Who indirectly calls f?
- How should f's costs be attributed?

Sharpening Calling Context

Calling context tree

- **Each node represents a PC location within a procedure**
- **Each edge represents a call site**
- **Call sites and PC locations are not unique within the tree**



Collecting Calling Context Trees

- Stack unwinding for dynamic context
  - —when a sample is taken, unwind call stack to see dynamic context
  - —can be difficult without frame pointer (e.g. Alpha + Compaq optimizer)
- Call counting
  - —at sample time
    - – use “mark bit” in stack frame (e.g. low order address bit) to know frame has been visited
  - —upon return
    - – use “trampoline” mechanism rather that triggers upon returns and records “call graph edge” traversal
- Advantages of these call counting mechanisms
  - —precisely attribute samples to calling context
  - —overhead proportional to sampling frequency rather than call frequency

Q-Tools

Designed for Itanium2

- Collect flat profile samples
- Collect “sampled call graph” edges by sampling “return” branch with HW performance counter
  - —at sample: receive PC (callee) and branch target (caller)
  - —not all dynamic call edges present: infrequent edges may be missed
- Presentation: qview
  - —infer inclusive and exclusive time using edge frequencies
- Advantages
  - —no compile-time support or dynamic instrumentation needed
  - —low overhead except when sampling
  - —stack unwinding unnecessary
  - —system-wide data collection (user processes and kernel)
  - —supports multi-threaded applications and shared libraries
- Disadvantages
  - —gprof-like assumptions about attribution

http://www.hpl.hp.com/research/linux/q-tools

System-wide Profiling

DCPI (Anderson et al. ACM TOCS '97)

- Collect samples in both user mode and kernel mode
  - —sample: (PID, PC, event)
- Profiles
  - —applications, kernel, device drivers, shared libraries, etc.
- Sampling interval
  - —avoid systematic errors: randomly select sampling interval between 60K and 64K cycles (uniform distribution)
- Very efficient!
  - —<1% for uniprocessor workloads
  - —-<5% overhead for multiprocessor workloads

Other system-wide profilers: Oprofile, Q-tools

Kernel Level Data Collection

- **Strategy**
  - —accumulate samples into histogram in kernel
    - – reduce data rate to user-level monitor
  - —place histogram overflow entries into output buffer
  - —dual output buffers for data streaming to user space
- **Designed for speed**
  - —design device driver structures to minimize cache misses
  - —allocate per processor data structures to avoid synch overhead
  - —switch among specialized versions of interrupt handler

User Level Daemon

- Waits until device driver signals an overflow
- Copies buffer from kernel and accumulates histograms
  - —maintains image map with each active process
    - – notified by loader whenever new image loaded, image path name
  - —maintains histograms (per image, per event type)
- Stores performance database on disk
  - —user-specified directory
  - —per image data
  - —samples organized into non-overlapping epochs
  - —binary format files

Data Analysis with DCPI

- **dcpiprof**
  - typical flat profiler: analyze samples per procedure or process image
- **dcpicalc**
  - calculates cycles per instruction
    - best case assuming statically-predictable stalls only (no cache miss)
    - infer stalls due to cache & TLB misses, slotting hazards,
  - calculates basic block execution frequencies
  - identifies possible causes for stalls
  - computes program-wide summary of effects of each kind of stalls
- **dcpistats**
  - analyzes variations in profile data from many runs
    - computes mean, min, max, std dev of procedure-level profiles