



Center for Scalable Application Development Software

# Sampling-based Strategies for Measurement and Analysis

John Mellor-Crummey  
Department of Computer Science  
Rice University





# Collaborators

---

- Rice
  - Nathan Tallent
  - Mark Krentel
  - Mike Fagan
  - Gabriel Marin
- Former students
  - Nathan Froyd
  - Cristian Coarfa
- RENCI
  - Rob Fowler



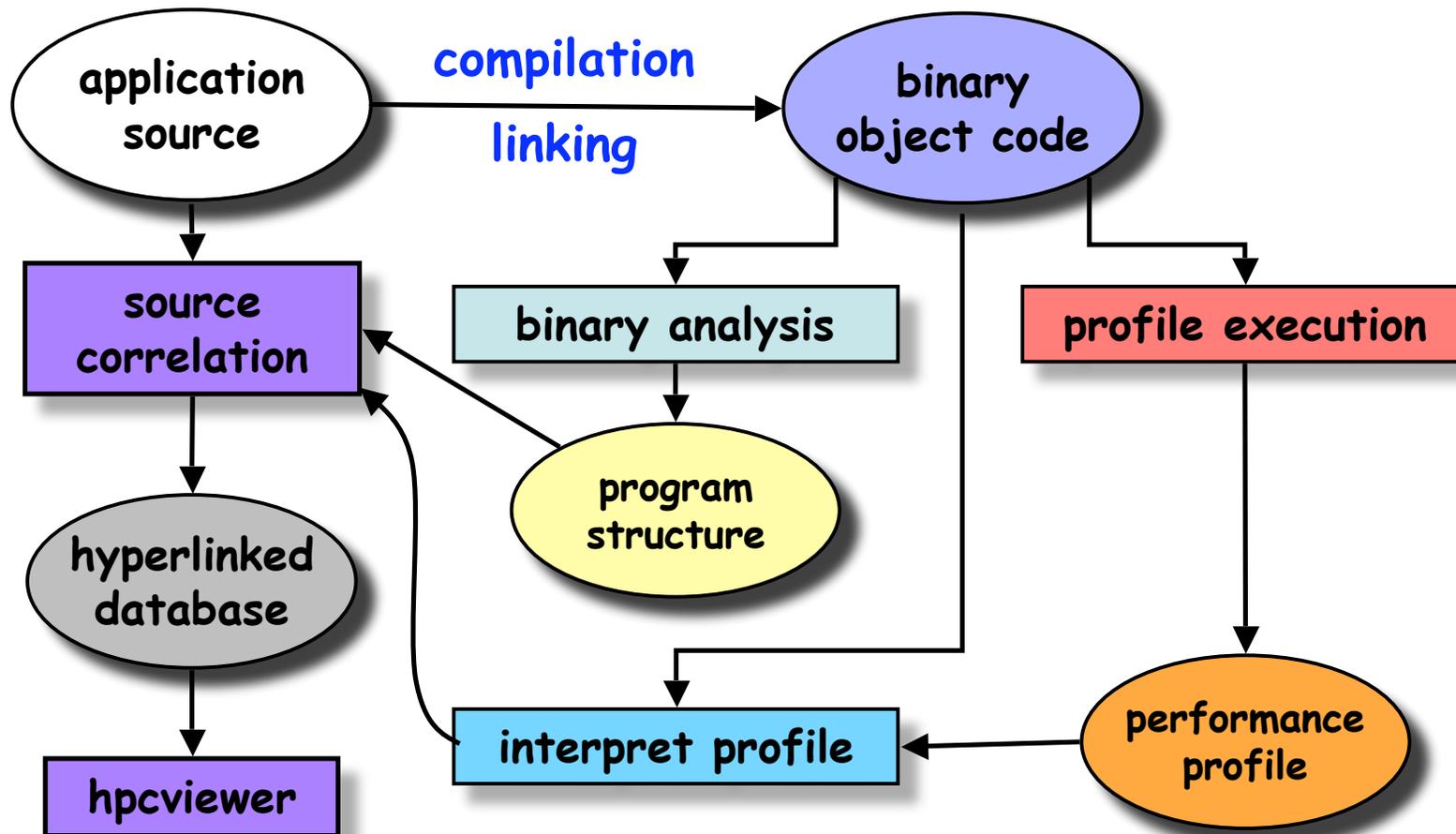
# Rice's HPCToolkit Philosophy

---

- Work at binary level for language independence
  - support multi-lingual codes with external binary-only libraries
- Profile rather than adding code instrumentation
  - minimize measurement overhead and distortion
  - enable data collection for large-scale parallelism
- Collect and correlate multiple performance measures
  - can't diagnose a problem with only one species of event
- Compute derived metrics to aid analysis
- Support top down performance analysis
  - intuitive enough for scientists and engineers to use
  - detailed enough to meet the needs of compiler writers
- Aggregate events for loops and procedures
  - accurate despite approximate event attribution from counters
  - loop-level info is more important than line-level info

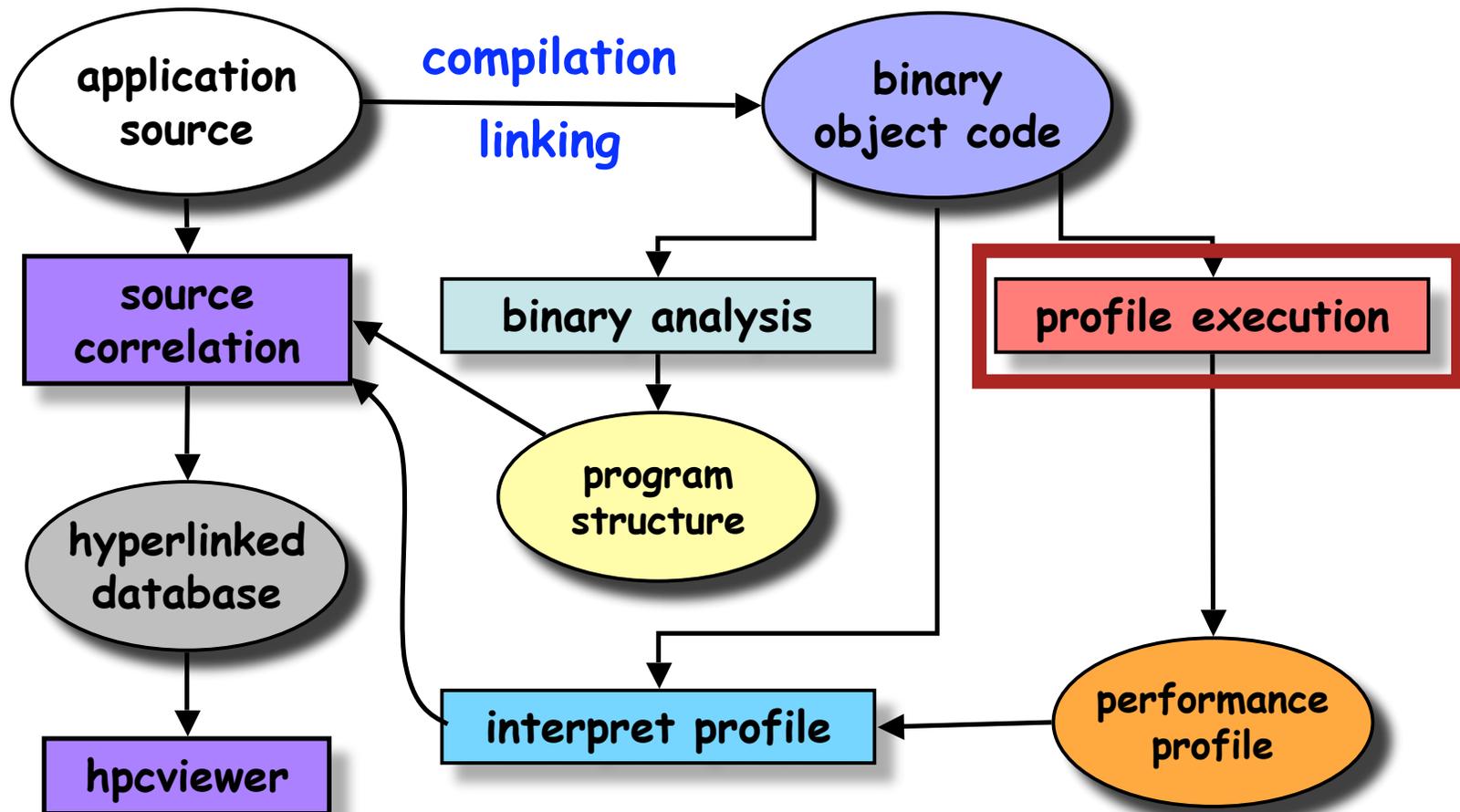


# HPCToolkit Workflow





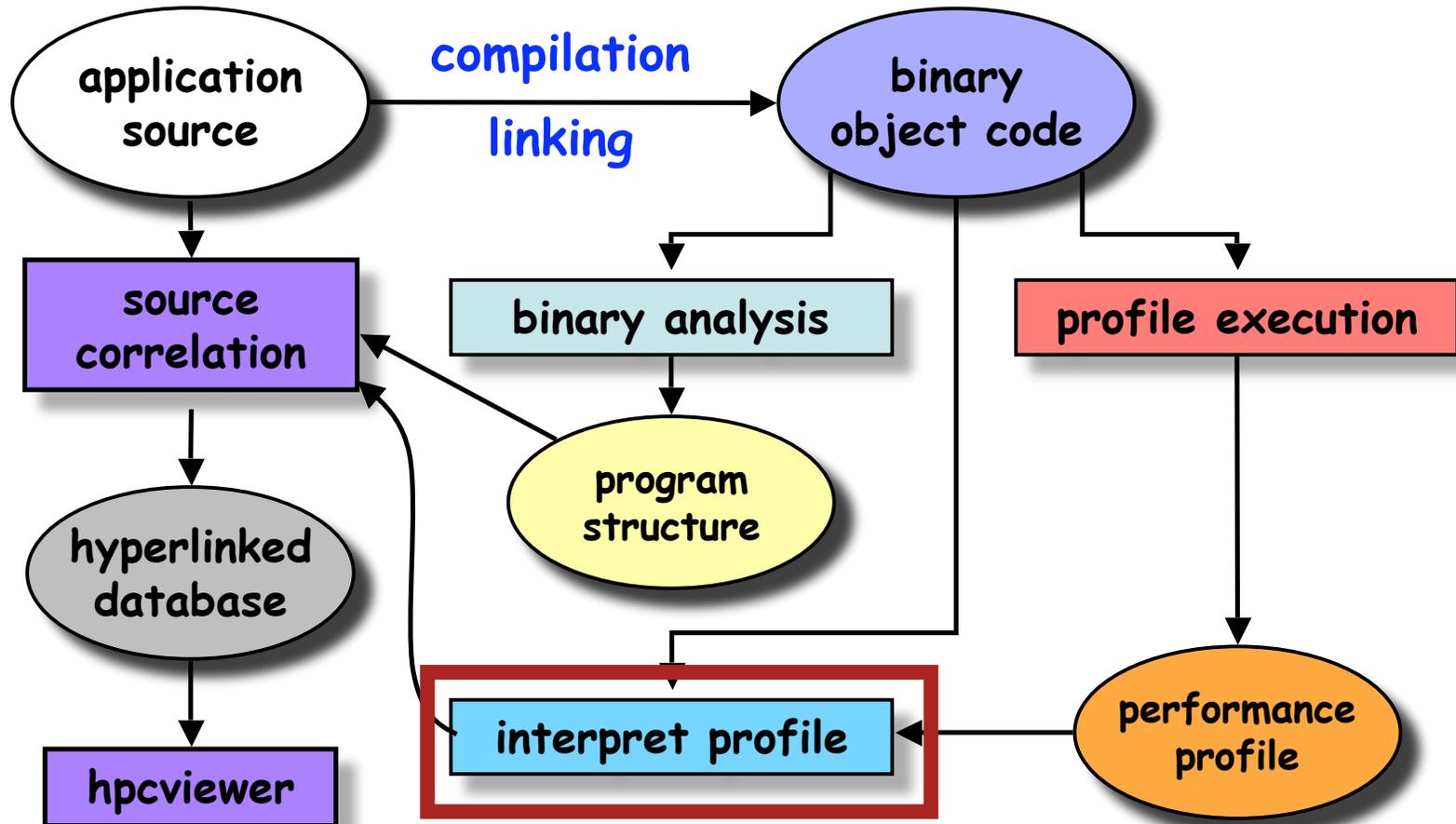
# HPCToolkit Workflow



- launch optimized application binaries
- collect statistical profiles of events of interest



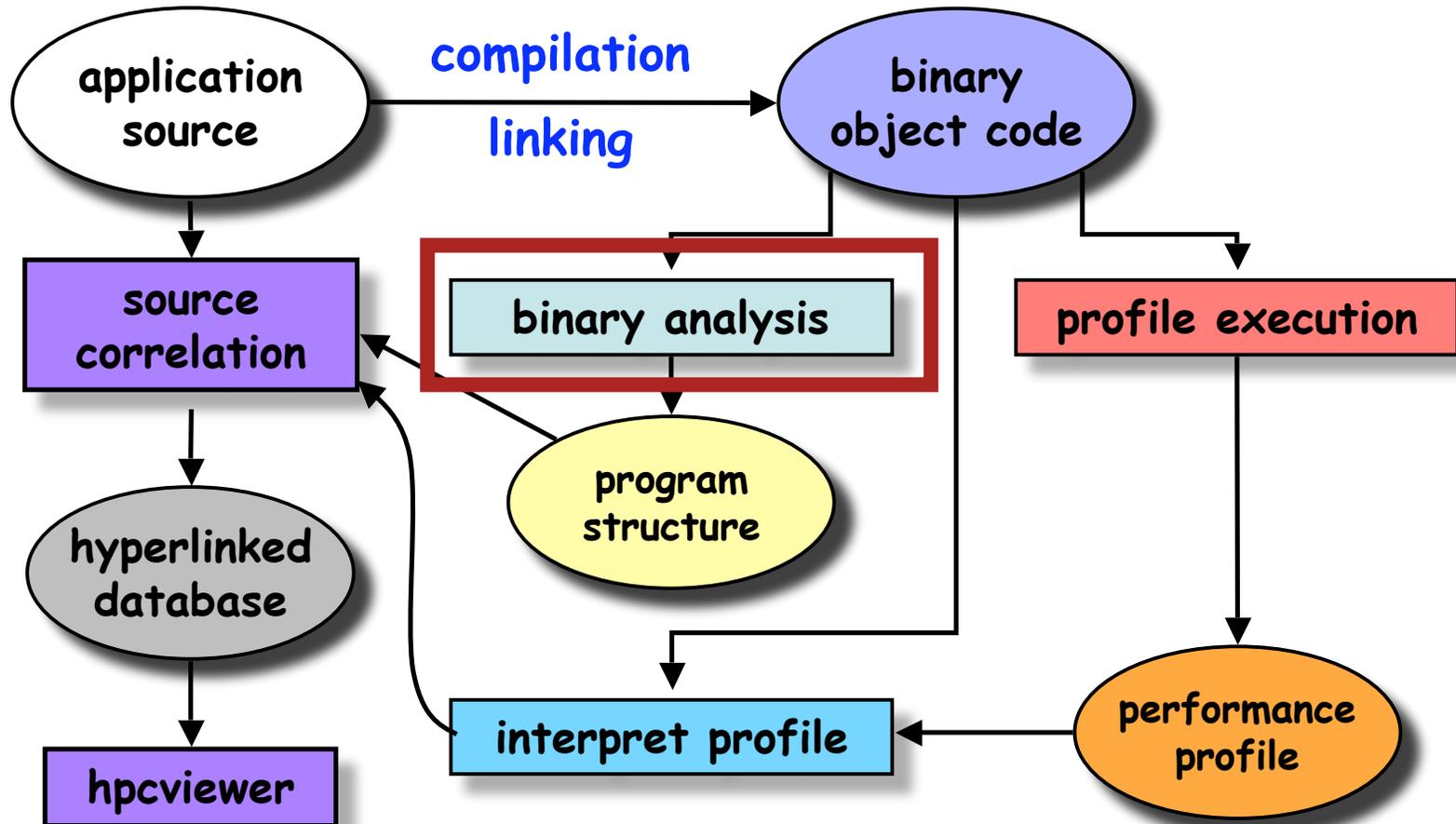
# HPCToolkit Workflow



- decode instructions and combine with profile data



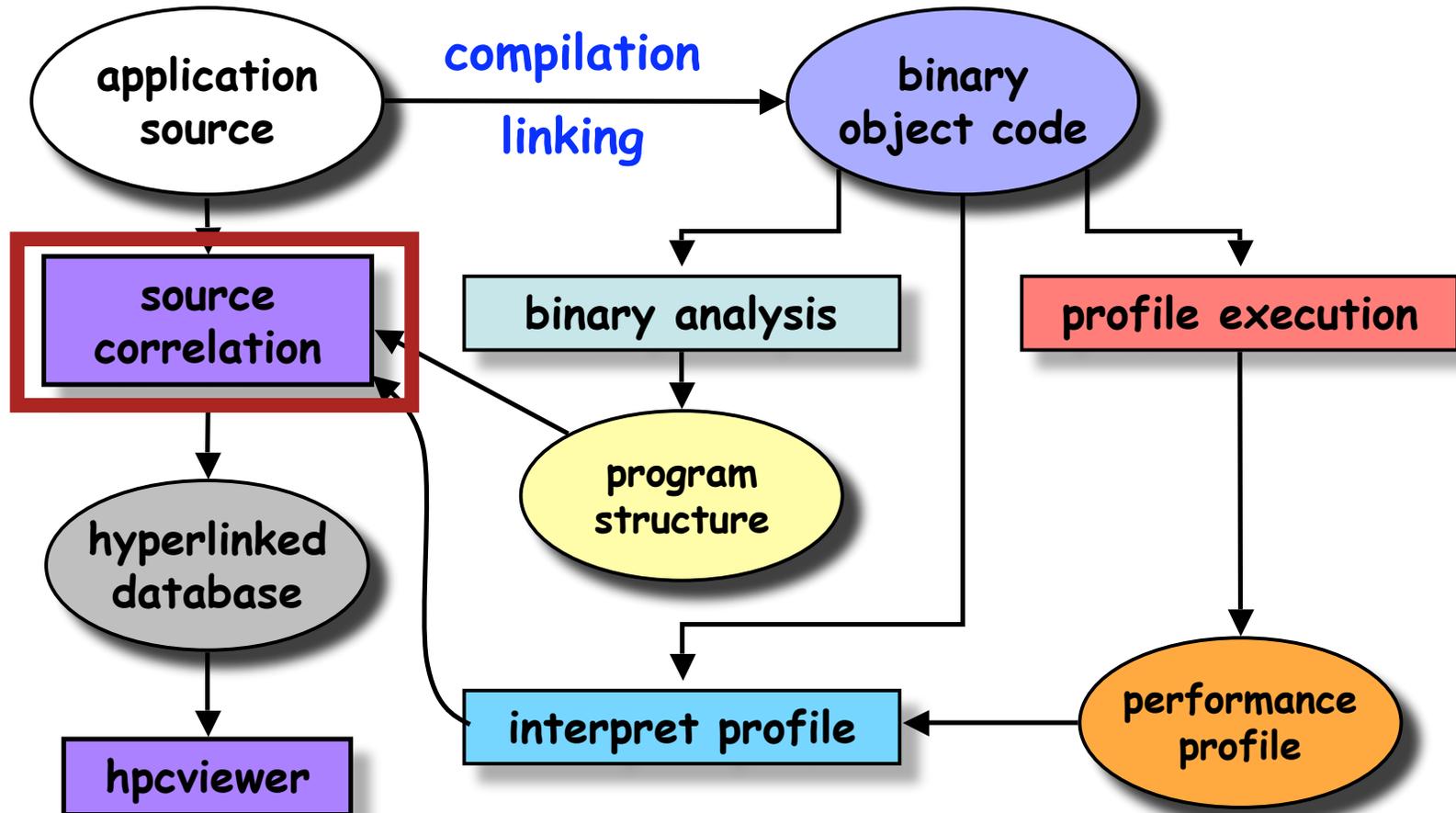
# HPCToolkit Workflow



- extract loop nesting & inlining from executables



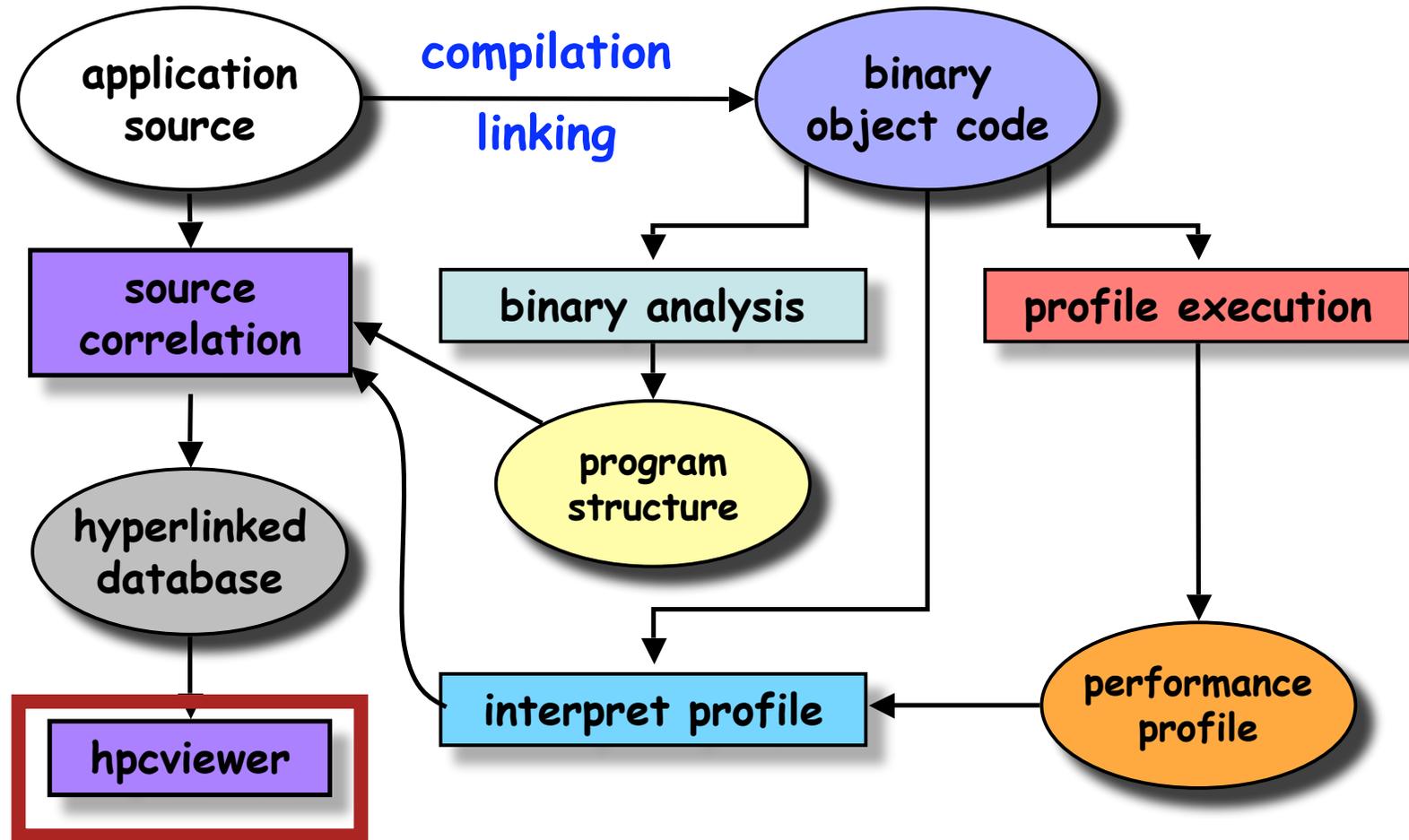
# HPCToolkit Workflow



- synthesize new metrics as functions of existing metrics
- relate metrics and structure to program source



# HPCToolkit Workflow



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



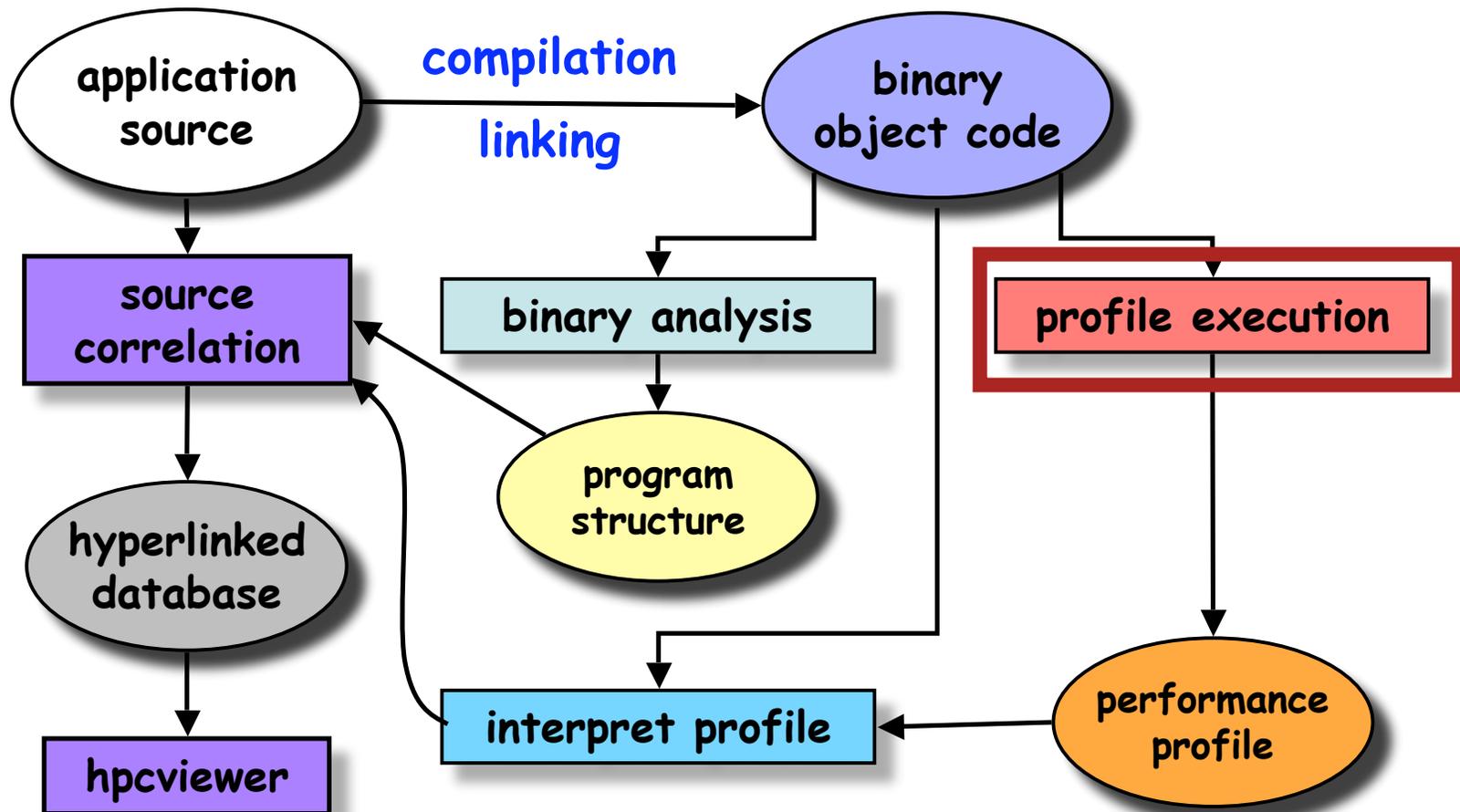
# Outline

---

- Sampling based measurement
- Binary analysis
- User interface
- Scalability analysis
- Components
  - ours
  - our desires
- Related modeling activities



# HPCToolkit Workflow





# Measurement Challenges

---

Performance often depends upon context

- Layered design
  - math libraries
  - communication libraries in parallel programs
- Generic programming, e.g. C++ templates
  - both data structures and algorithms
- Goals
  - identify and quantify context-sensitive behavior
  - differentiate between types of performance problems
    - cheap procedure called many times
    - expensive procedure called few times

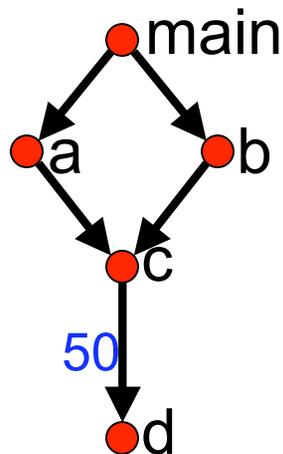


# Understanding Costs In Context

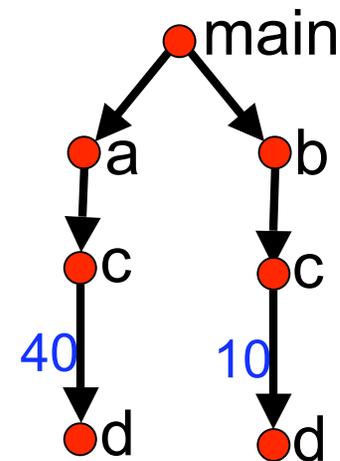
## Call Path Profiling

- Measure time spent in each procedure
- Attribute time upward along call chain
- Report average time per call per calling context

Call  
Graph



Calling  
Context  
Tree





# A Torture Test

```
#define HUGE (1<<28)
```

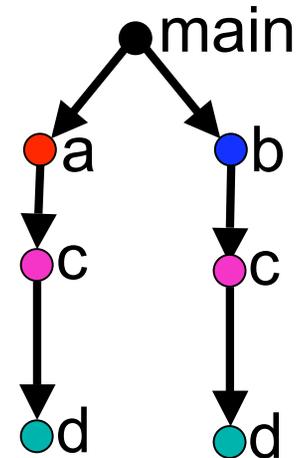
```
void d() {}
```

```
void c(long n) {  
    for(int j=0; j<HUGE/n; j++) d();  
}
```

```
void a(void (*f)(long)) { f(1); f(1); }
```

```
void b(void (*f)(long)) { f(2); f(2); f(2); f(2); }
```

```
void main() { a(c); b(c); }
```





# Results with Existing Tools

---

(for the torture test)

- Instrumentation-based profilers
  - gprof: dilates execution by a factor of 3-14
    - cannot distinguish different costs for calling contexts
  - Vtune: dilates execution by a factor of 31 (Linux+P4)!
- Call stack sampling profilers
  - e.g., Apple's Shark, HP's scgprof
    - can't distinguish different costs for calling contexts

csprof: 1.5% overhead; accurate context-based attribution



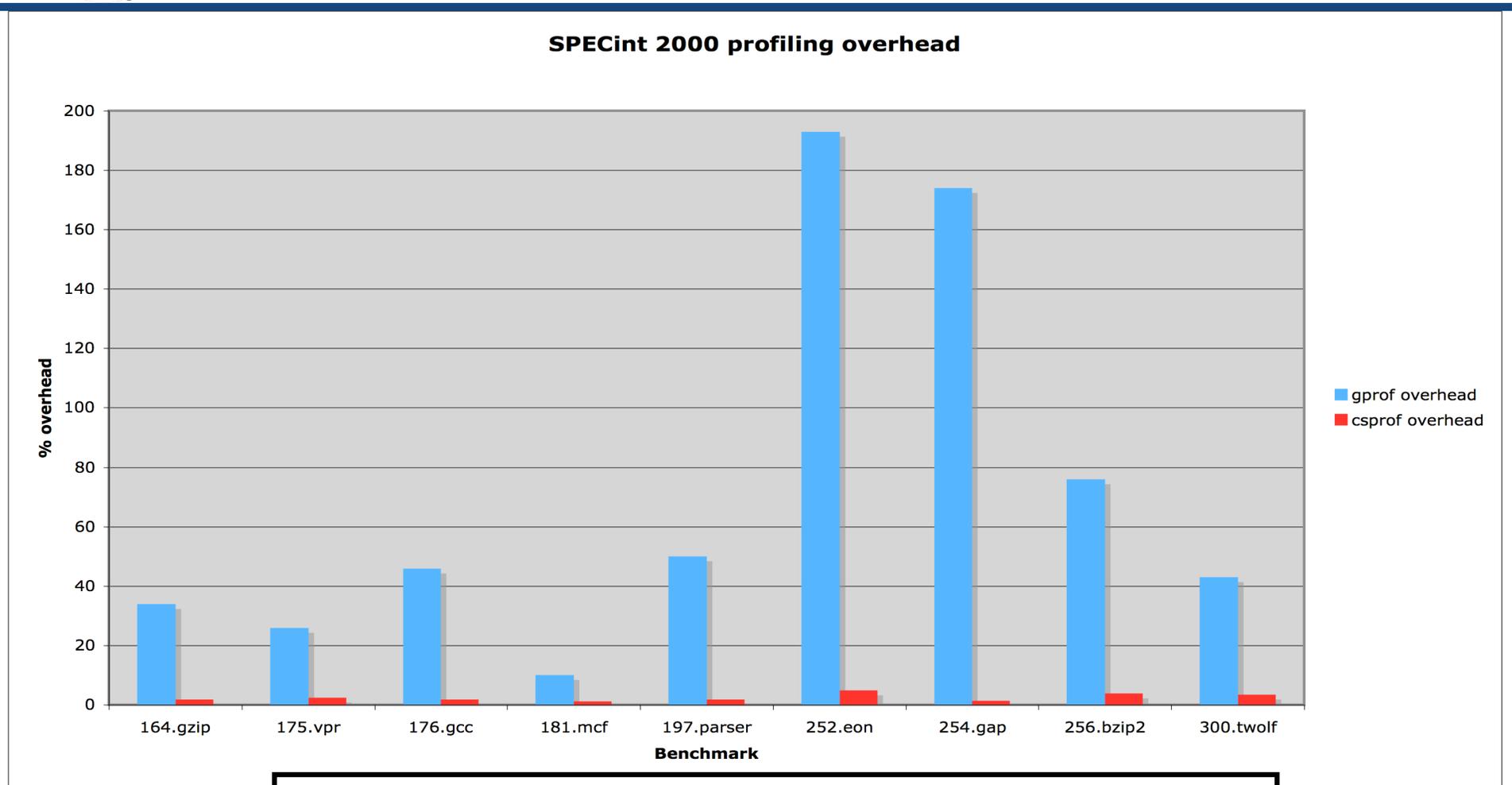
# Call Path Profiling Overview

---

- At each sample event
  - use call stack unwinding to identify full context
    - [vector of return addresses; PC]
  - record sample in a calling context tree (CCT)
    - captures common context between samples
  - “mark the current procedure frame”
    - replace frame’s return address with address of a “trampoline”
  - remember CCT path to marked frame
- When returning from a marked procedure frame
  - increment edge count of the last call edge in the memoized path
  - pop the last edge in the memoized path
  - mark the caller’s frame with the trampoline
  - return control to caller
- Low-overhead unwinding: need not unwind beyond marked frame



# SPECint 2000 Benchmarks

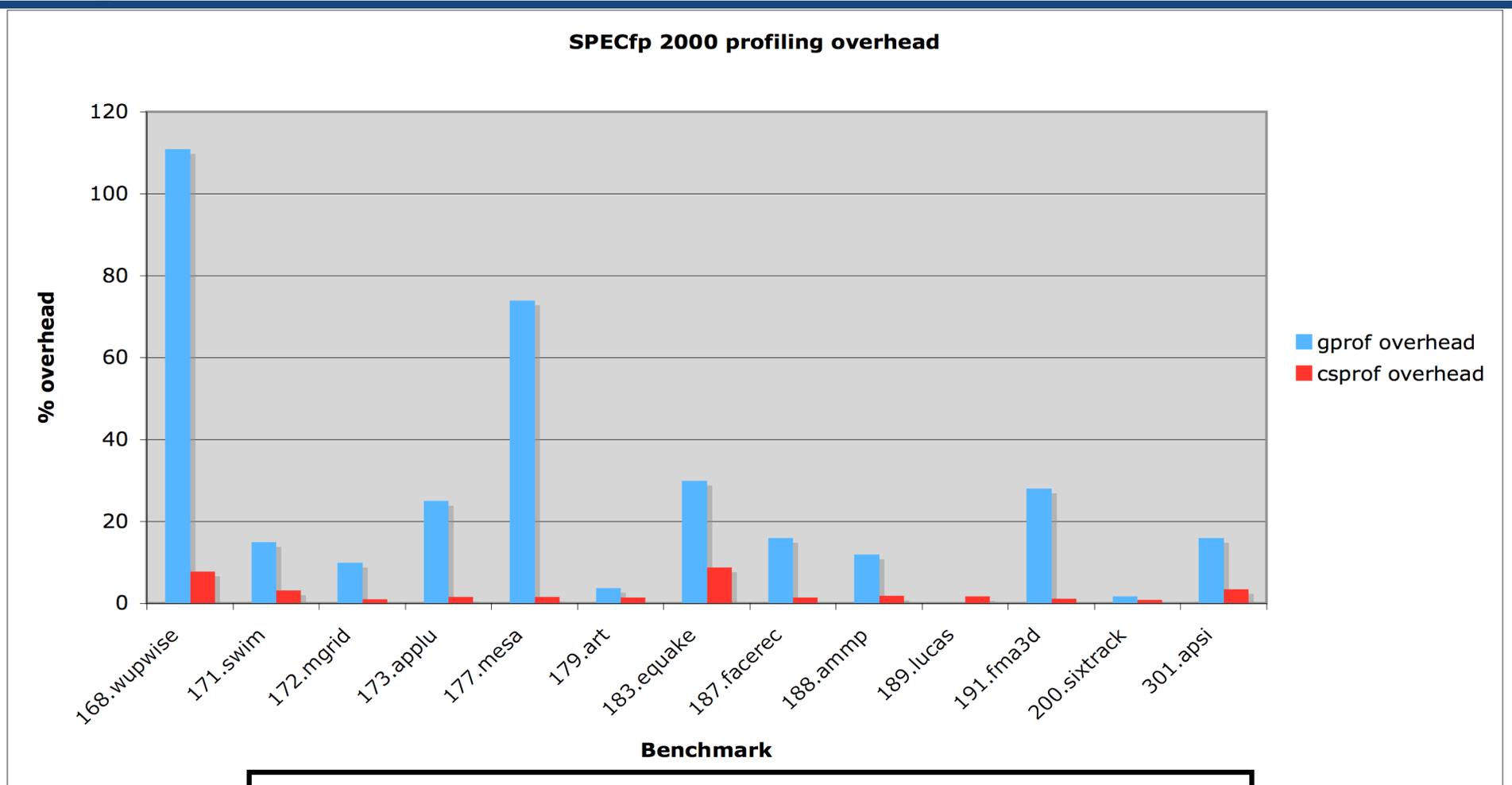


Average overhead: gprof 82%, csprof 2.7%

(Opteron, gcc 4.1)



# SPECfp 2000 Benchmarks



**Average overhead: gprof 31%, csprof 3.2%**

(Opteron, gcc 4.1)



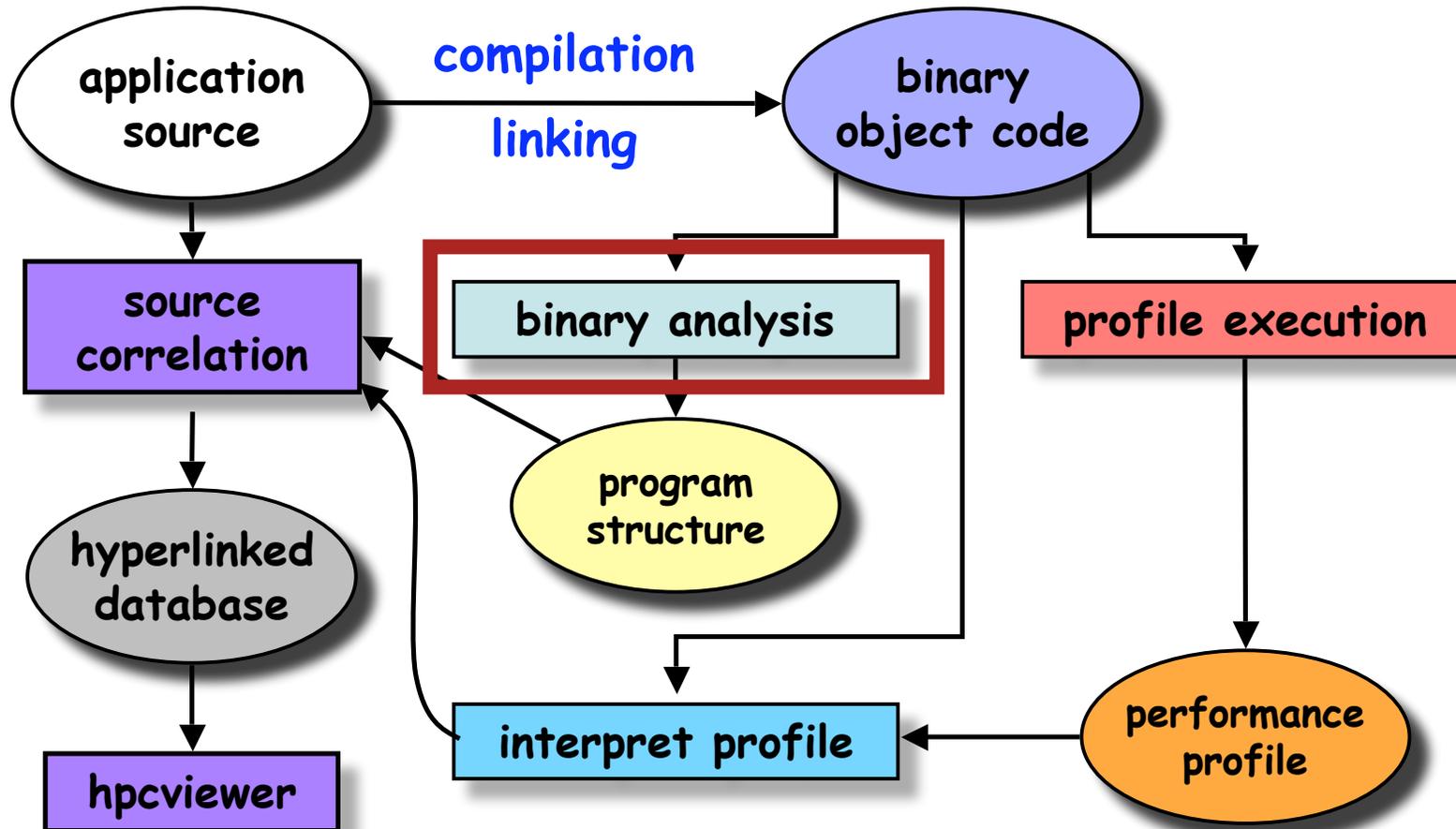
# Ongoing Call Path Profiler Refactoring

---

- Platform: OS, architecture
- Profiling flavor
  - flat vs. calling context (CC)
    - CC: precise vs. summary
    - CC: naive vs. smart unwinding (SU)
      - SU: compiler information vs. binary analysis (BA) vs. emulation
        - BA: eager vs. lazy
      - SU: edge counting vs. pure call stack sampling
    - threaded vs. non-threaded
- Initiation: preloading vs. static vs. attaching
- Synchronous vs. asynchronous events
- Asynchronous sample sources
  - timers, counters
  - instruction-based sampling
- Online control API



# HPCToolkit Workflow





# Why Binary Analysis?

---

- Understanding a program's performance requires understanding its structure
- Program structure after optimization may only vaguely resemble the program source
  - complex patterns of code composition
    - e.g. C++ expression templates
  - understanding loops is important to for understanding performance
    - account for significant time in data-intensive scientific codes
    - undergo significant compiler transformations

Goal: understand transformed loops in the context of transformed routines



# Program Structure Recovery with **bloop**

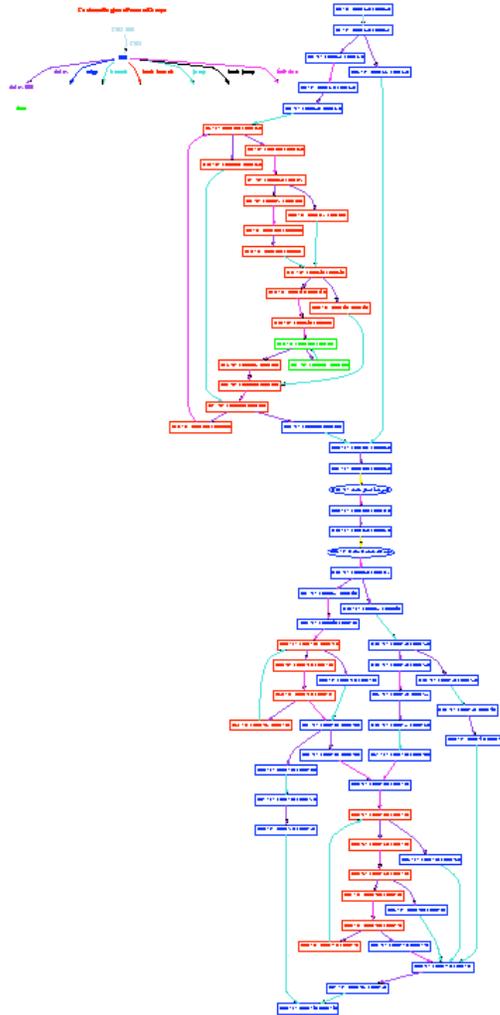
---

## Analyze an application binary

- Construct control flow graph from branches
- Identify natural loop nests using interval analysis
- Map instructions to source lines, procedures
  - leverage line map + DWARF debugging information
- Recover procedure boundaries
- Identify inlined code & its nesting in procedures and loops
- Normalize loop structure information to recover source-level view



# Sample Flowgraph from an Executable



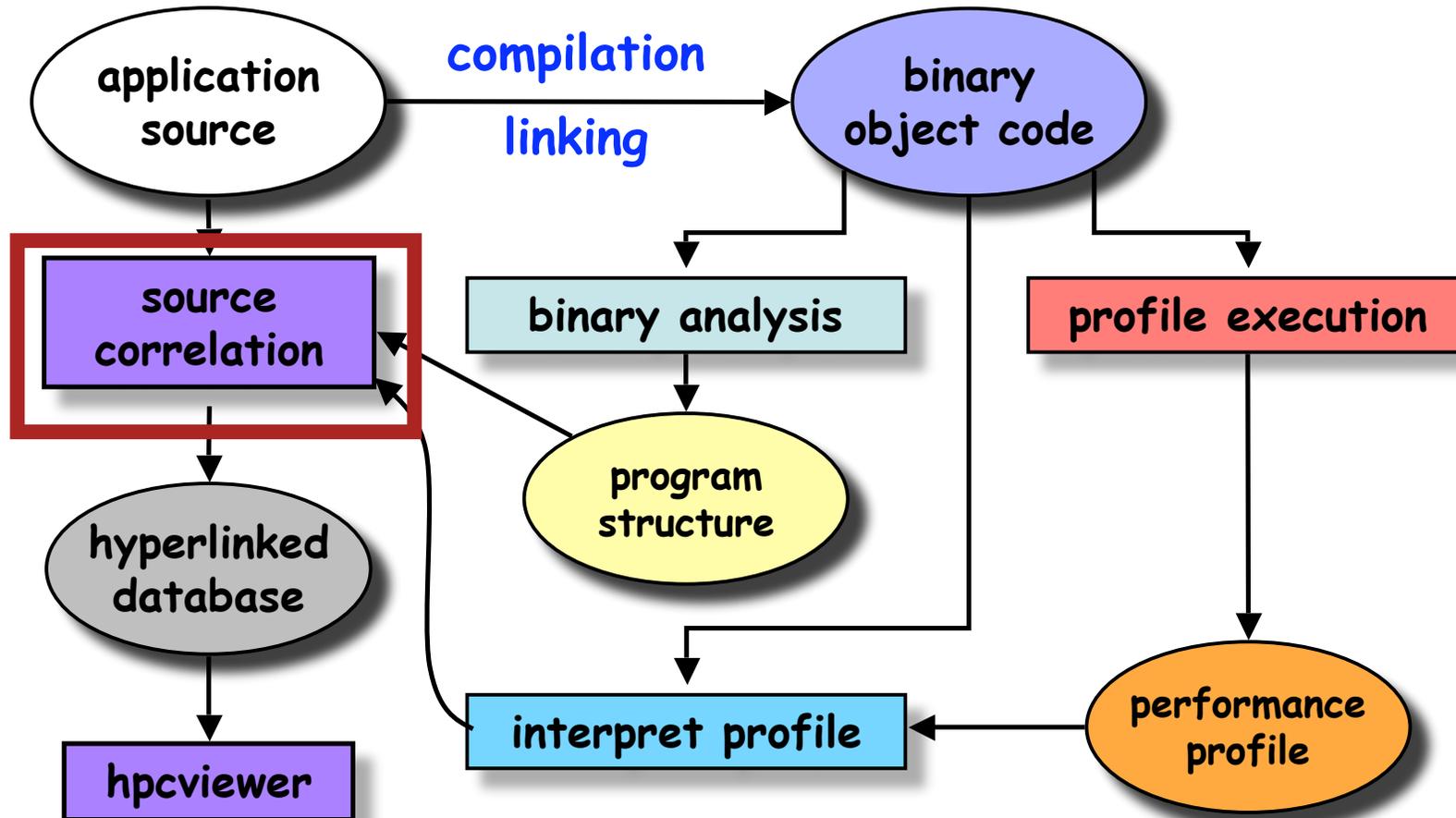
Loop nesting structure

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

**Observation**  
**optimization complicates**  
**program structure!**



# HPCToolkit Workflow





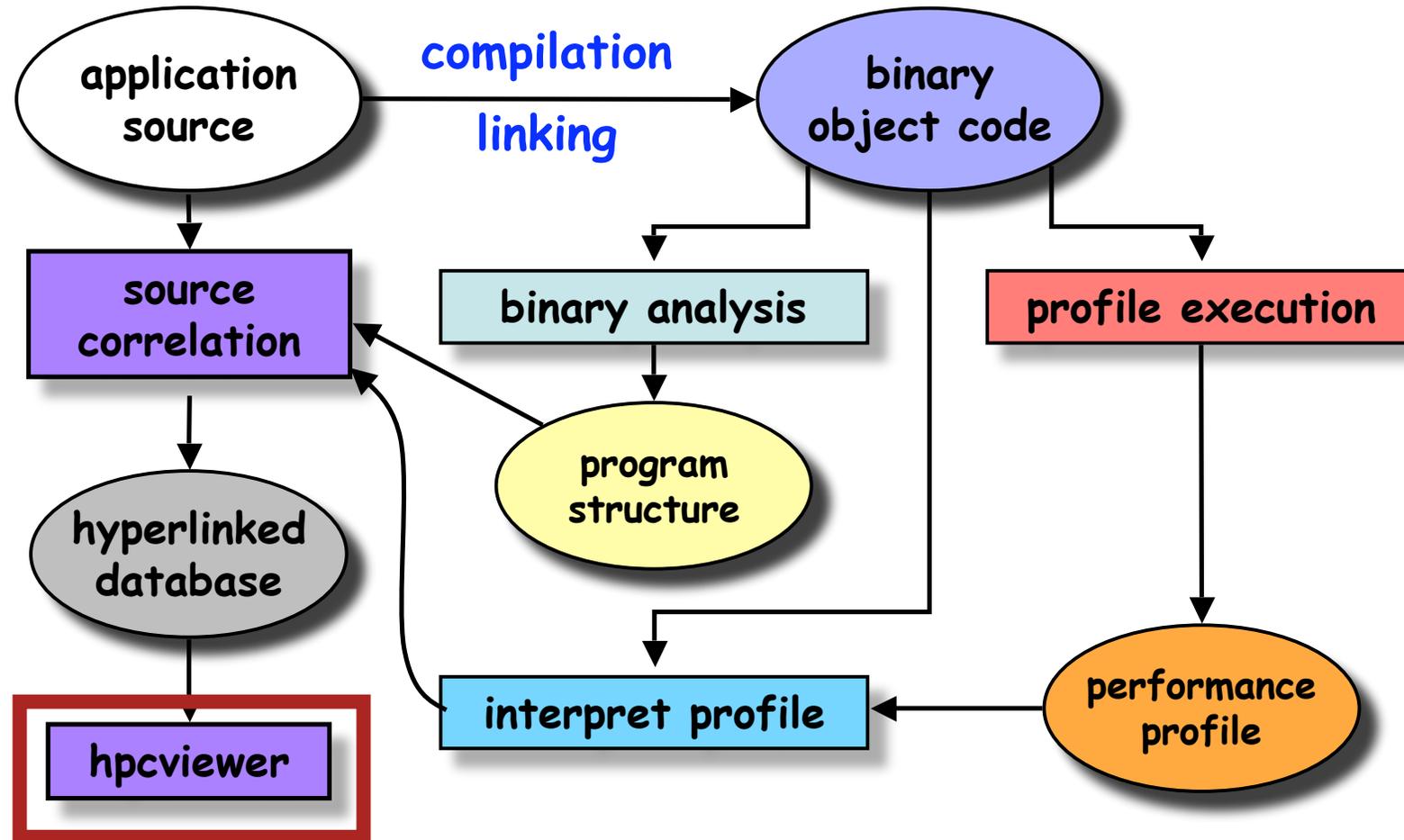
# Data Correlation

---

- Problem
  - any one performance measure provides a myopic view
    - some measure potential causes (e.g. cache misses)
    - some measure effects (e.g. cycles)
    - cache misses not always a problem
  - event counter attribution is often inaccurate
- Approaches
  - multiple metrics for each program line
  - computed metrics, e.g. peak FLOPs - actual FLOPS
    - eliminates mental arithmetic
    - serves as a key for sorting
  - hierarchical structure
    - errors with line level attribution still yield good loop-level information



# HPCToolkit System Overview





# hpcviewer User Interface

The screenshot shows the hpcviewer interface with the following components highlighted:

- source pane:** The top window showing the source code of `hmc.cc`. The `main` function is highlighted.
- view control:** A control bar with buttons for `Calling Context View`, `Callers View`, and `Flat View`.
- flatten/zoom control:** A control bar with a `Scopes` label and icons for zooming in and out.
- navigation pane:** A tree view showing the execution context, including `main`, `void Chroma::doHMC`, and various nested loops and function calls.
- metric pane:** A table showing performance metrics for different execution contexts.

# samples (I)	# samples (E)
5.80e05	97.5%
5.70e05	95.8%
5.65e05	95.0%
5.65e05	95.0%
5.25e05	88.2%
4.25e05	71.4%
3.30e05	55.5%



# hpcviewer Views

---

- Calling context view
  - top-down view shows dynamic calling contexts in which costs were incurred
- Caller's view
  - bottom-up view apportions costs incurred in a routine to the routine's dynamic calling contexts
- Flat view
  - aggregates all costs incurred by a routine in any context and shows the details of where they were incurred within the routine



# Calling Context View: Chroma Lattice QCD

```
qdp_parscalar_specific.h
77 {
78   QMP_sum_float_array(dest, len);
79 }
80
81  //!< Low level hook to QMP_global_sum
82  inline void globalSumArray(double *dest, int len)
83  {
84  QMP_sum_double_array(dest, len);
85  }
86
87  //!< Global sum on a multi1d
88  template<class T>
89  inline void globalSumArray(multi1d<T>& dest)
90  {
```

static + dynamic structure

- costs for loops in CCT
- costs for inlined procedures
- inclusive and exclusive costs

Calling Context View Callers View Flat View

Scopes

	# samples (I)	%	# samples (E)	%
Chroma::TwoFlavorExactWilsonTypeFermMonomial<QDP::multi1d<QDP::OLattice<QDP::PScalar<QDP::PColorMatrix>>>	2.30e05	38.7%		
Chroma::MdagMSysSolverCG<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	2.20e05	37.0%		
loop at sysolver_mdagm_cg.h: 66-70	2.20e05	37.0%		
Chroma::SystemSolverResults_t Chroma::InvCG2<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	2.20e05	37.0%		
Chroma::SystemSolverResults_t Chroma::InvCG2_a<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	2.20e05	37.0%	1.00e04	1.7%
loop at invcg2.cc: 147-182	1.85e05	31.1%	5.00e03	0.8%
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	1.05e05	17.6%	1.00e04	1.7%
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	7.00e04	11.8%	1.50e04	2.5%
[I] globalSumArray	5.00e03	0.8%		
[I] vaxpy3	5.00e03	0.8%	5.00e03	0.8%
[I] local_sumsq				
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	2.00e04	3.4%		
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>>>	5.00e03	0.8%		



# Fusing Static + Dynamic Structure: Chroma

The screenshot displays a call stack for a function named 'main'. Several loops are highlighted with red boxes, indicating their execution locations and line ranges:

- loop at hmc.cc: 311-435
- loop at lcm\_sts\_leapfrog\_recursive.cc: 129-131
- loop at lcm\_exp\_sdt.cc: 85-94
- loop at sysssolver\_mdagm\_cg.h: 66-70
- loop at invcg2.cc: 147-207
- loop at qdp\_generic\_fused\_spin\_proj\_evaluates.h: 162-161

On the right side of the call stack, a performance table is visible, showing execution time and percentage for various routines:

Chroma::TwoFlavorExactWilsonTypeFermMonomial<QDP::multi1d<QDP::OLattice<QDP::PScalar<QDP::PCol...>>>	7.68e06	69.2%
Chroma::TwoFlavorExactWilsonTypeFermMonomial<QDP::multi1d<QDP::OLattice<QDP::PScalar<QDP::PCol...>>>	6.15e06	55.3%
Chroma::MdagMSysSolverCG<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	6.06e06	54.5%
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	5.25e06	47.3%
Chroma::QDPWilsonDslashOpt::apply(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	2.54e06	22.9%
Chroma::QDPWilsonDslashOpt::apply(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	1.27e06	11.4%
void QDP::OSubLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	1.60e05	1.4%
void QDP::OSubLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>	1.60e05	1.4%
void QDP::evaluate<QDP::PSpinVector<QDP::PColorVector<QDP::RC...>>>		
loop at qdp_generic_fused_spin_proj_evaluates.h: 162-161		
qdp_generic_fused_spin_proj_evaluates.h: 162		
qdp_generic_fused_spin_proj_evaluates.h: 161		

Three callouts in red rounded rectangles provide additional context:

- Dynamically nested loops**
- Routines marked inline, not inlined**
- Inlined routines**

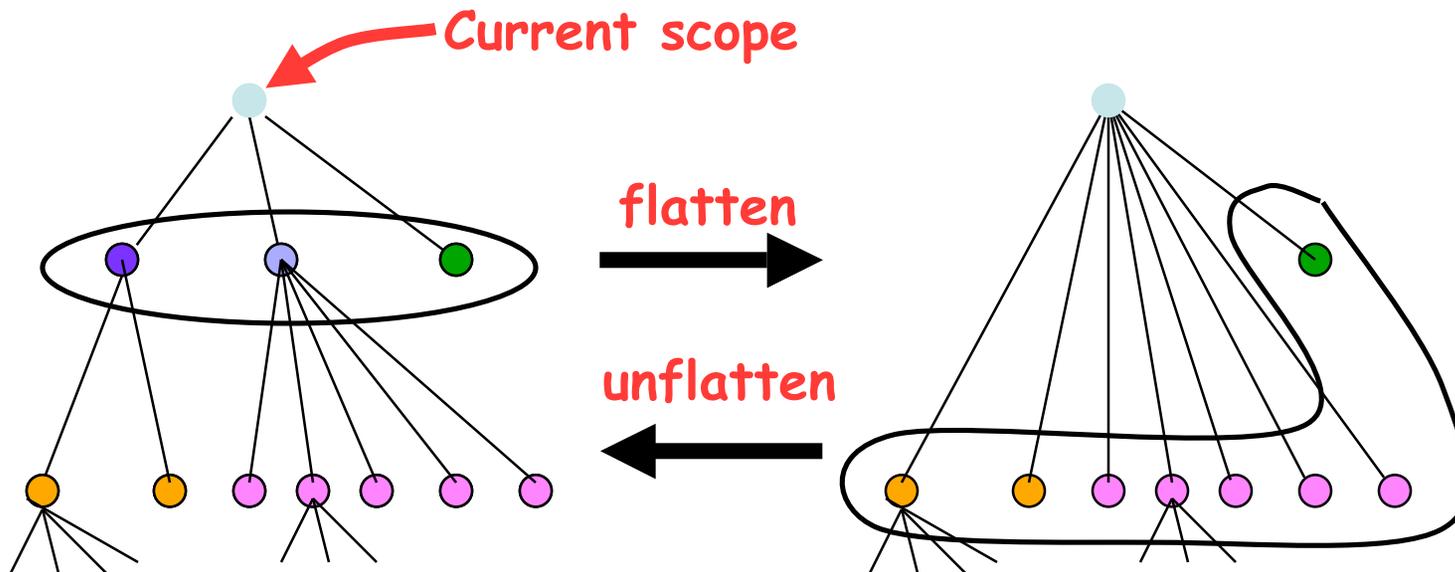
A fourth callout in a red rounded rectangle at the bottom right states: **6 loops around operator evaluations**





# Flattening Static Hierarchies

- Problem
  - hierarchical view of a program is too rigid
  - sometimes want to compare children of different parents
    - e.g. compare all loops, regardless of the routine they are inside
- Solution
  - flattening elides a scope and shows its children instead





# Flat View: S3D Combustion Code

```
734 diffFlux(:,:,:,n_spec,:) = 0.0
735 DIRECTION: do m=1,3
736 SPECIES: do n=1,n_spec-1
737
738 if (baro_switch) then
739 ! driving force includes gradient in mole fraction and baro-diffusion:
740 diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m) &
741 + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m) &
742 + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Pre: ...
743 else
744 ! driving force is just the gradient in mole fraction:
745 diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)
746 + Ys(:,:,:,n) * grad_mixMW(:,:,:,m) )
747 endif
748
749 ! Add thermal diffusion:
750 if (thermDiff_switch) then
751 diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m) &
752 - Ds_mixavg(:,:,:,n) * Rs_therm_diff(:,:,:,n) * molwt(n) &
753 * avmolwt * grad_T(:,:,:,m) / Temp
754 endif
755
```

Calling Context View Callers View **Flat View**

Scopes

- loop at mixavg\_transport\_m.f90: 735-760
  - loop at mixavg\_transport\_m.f90: 736-758
    - loop at mixavg\_transport\_m.f90: 745**
    - loop at mixavg\_transport\_m.f90: 758
    - loop at mixavg\_transport\_m.f90: 740
    - loop at mixava transport m.f90: 751

# samples (I)		# samples (E)	
2.17e07	11.3%	2.17e07	11.3%
2.17e07	11.3%	2.17e07	11.3%
<b>1.54e07</b>	<b>8.0%</b>	<b>1.54e07</b>	<b>8.0</b>
6.32e06	3.3%	6.32e06	3.3

attribute costs to loops  
implicit with F90 vector syntax

fine-grain attribution to loops  
within a loop nest



# Another Flat View of S3D

```
199 ! grad_Y - Species mass fraction gradients may be required in transport
200 !     evaluation as well as for boundary conditions.
201 !
202 !notes by ramanan - 01/05/05
203 !The array dimensioning can be misleading
204 !For grad_u, 4th dimension is the direction and 5th dimension is the velocity component
205 !For grad_Ys, 4th dimension is the species and 5th dimension is the direction
206
207 call computeVectorGradient( u, grad_u )
208 call computeScalarGradient( temp, grad_T )
209 do n=1,n_spec
210   call computeScalarGradient( yspecies(:,:,:,n), grad_Ys(:,:,:,n) )
211 enddo
212
213 !Added by Ramanan - 01/05/05
214 !Store the boundary grad values
215 if(vary_in_x==1)then
216   if (xid==0) then
217     grad_u_x0 = grad_u(1,:,:,1,:)
218     grad_Ys_x0 = grad_Ys(1,:,:,1)
219     h_spec_x0 = h_spec(1,:,:,)
220   end if
```

Calling Context View Callers View **Flat View**

Scopes		# samples (I)	# samples (E)
Experiment Aggregate Metrics		1.91e08 100.0	1.91e08 100.0
~~~s3d_f90.x: <unknown-file>~~~: 0		2.60e07 13.6%	2.60e07 13.6%
▶ loop at mixavg_transport_m.f90: 735-760		2.17e07 11.3%	2.17e07 11.3%
▼ loop at rhsf.f90: 209-210		2.03e07 10.6%	1.04e07 5.4%
▶ loop at rhsf.f90: 210		2.03e07 10.6%	1.04e07 5.4%
▶ loop at mixavg_transport_m.f90: 1004-1011		8.94e06 4.7%	8.94e06 4.7%

highlights costs for an implicit loop that copies non-contiguous 4D slice of 5D data to contiguous storage



# Computed Metrics for S3D

/Users/johnmc/Documents/Admin/Grants/Active/DOE/PERI/Tiger Teams/S3D/s3d-opteron-1cpu-20iterations-hpctoolkit-db...

mixavg\_transport\_m.f90

```
734 diffFlux(:,:,:,n_spec,:) = 0.0
735 DIRECTION: do m=1,3
736 SPECIES: do n=1,n_spec-1
737
738 if (baro_switch) then
739 ! driving force includes gradient in mole fraction and baro-diffusion:
740 diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m) &
741 + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m) &
742 + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Press))
743 else
744 ! driving force is just the gradient in mole fraction:
745 diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m) &
746 + Ys(:,:,:,n) * grad_mixMW(:,:,:,m) )
747 endif
748
749 ! Add thermal diffusion:
750 if (thermDiff_switch) then
751 diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m) &
752 - Ds_mixavg(:,:,:,n) * Rs_therm_diff(:,:,:,n) * molwt(n) &
753 * avmolwt * grad_T(:,:,:,m) / Temp
754 endif
755
756 ! compute contribution to nth species diffusive flux
757 ! this will ensure that the sum of the diffusive fluxes is zero.
758 diffFlux(:,:,:,n_spec,m) = diffFlux(:,:,:,n_spec,m) - diffFlux(:,:,:,n,m)
759
760 enddo SPECIES
761 enddo DIRECTION
```

**Overall performance (15% of peak)**  
 **$2.05 \times 10^{11}$  FLOPs /  $6.73 \times 10^{11}$  cycles = .305 FLOPs/cycle**

**highlighted loop accounts for 11.4% of total program waste**

Flat View

Scopes	PAPI_TOT_CYC	PAPI_FP_INS	PAPI_TOT_INS	PAPI_STL_ICY	WASTE
Experiment Aggregate Metrics					
loop at mixavg_transport_m.f90: 735-760	6.73e11 100.0	2.05e11 100.0	4.56e11 100.0	1.59e10 100.0	1.14e12 100.0
loop at mixavg_transport_m.f90: 736-760	0.90e10 10.3%	9.00e09 4.4%	4.06e10 8.9%	1.32e09 8.3%	1.30e11 11.4%
loop at mixavg_transport_m.f90: 735	6.96e10 10.3%	9.00e09 4.4%	4.06e10 8.9%	1.32e09 8.3%	1.30e11 11.4%
loop at rhsf.f90: 209-210	1.00e06 0.0%				2.00e06 0.0%
loop at rhsf.f90: 209-210	7.58e10 11.3%	4.23e10 20.6%	8.29e10 18.2%	1.07e09 6.7%	1.09e11 9.6%
loop at mixavg_transport_m.f90: 908-914	3.88e10 5.8%		1.79e10 3.9%	4.24e08 2.7%	7.75e10 6.8%
loop at mixavg_transport_m.f90: 908-914	3.19e10 4.7%	4.41e09 2.1%	1.46e10 3.2%	4.80e08 3.0%	5.95e10 5.2%

**Wasted Opportunity (Maximum FLOP rate \* cycles - (actual FLOPs))**



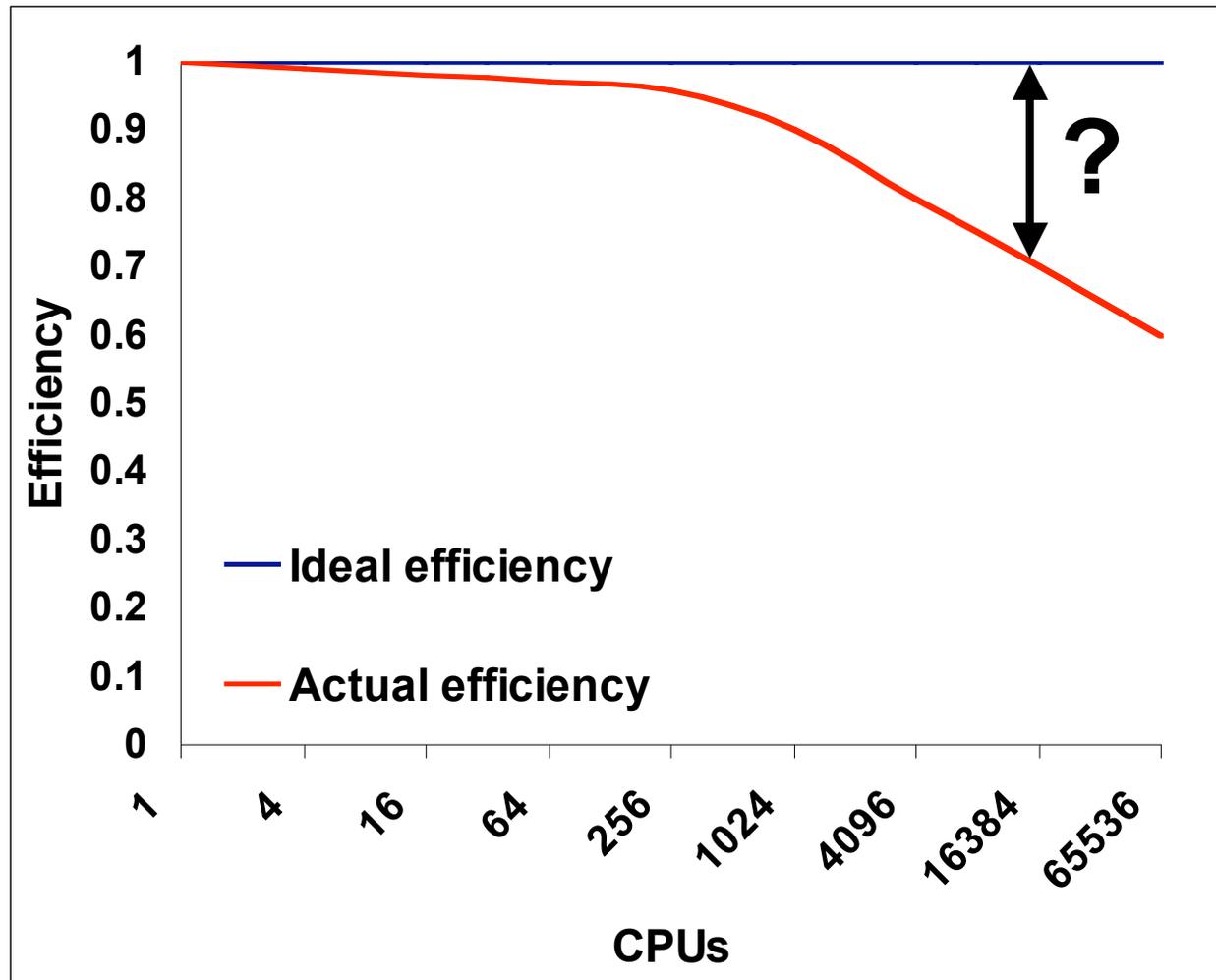
# Outline

---

- Sampling based measurement
- Binary analysis
- User interface
-  Scalability analysis
- Components
  - ours
  - our desires
- Related modeling activities



# The Lump Under the Rug: Scaling Bottlenecks



Note: higher is better

Synthetic Example



# Impediments to Scalability

---

- Communication overhead
  - synchronization
  - data movement
- Computation overhead
  - replicated initialization
  - partially replicated computation
- Parallelization deficiencies
  - load imbalance
  - serialization
- Algorithmic scaling
  - e.g. reductions: time increases as  $O(\log P)$



# Goal: Automatic Scaling Analysis

---

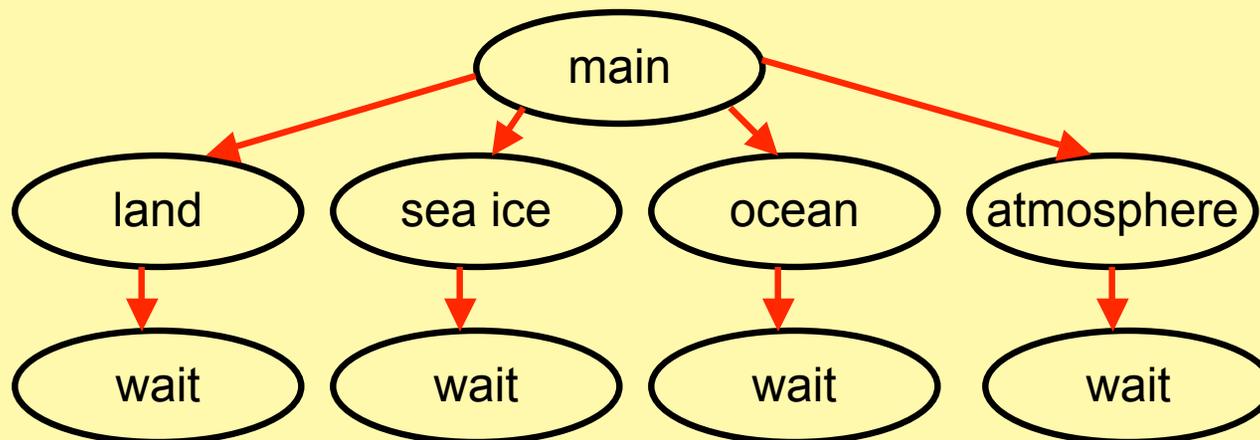
- Pinpoint scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem
- Diagnose the nature of the problem



# Challenges for Pinpointing Scalability Bottlenecks

- Parallel applications
  - modern software uses layers of libraries
  - performance is often context dependent
- Monitoring
  - bottleneck nature: computation, data movement, synchronization?
  - size of petascale platforms demands acceptable data volume
  - low perturbation for use in production runs

## Example climate code skeleton





# Performance Analysis with Expectations

---

- Users have performance expectations for parallel codes
  - strong scaling: linear speedup
  - weak scaling: constant execution time
- Putting expectations to work
  - define our expectations
  - measure performance under different conditions
    - e.g. different levels of parallelism or different inputs
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively



# Weak Scaling Analysis for SPMD Codes

## Performance expectation for weak scaling

- work increases linearly with # processors
- execution time is same as that on a single processor

- Execute code on  $p$  and  $q$  processors; without loss of generality,  $p < q$
- Let  $T_i$  = total execution time on  $i$  processors
- For corresponding nodes  $n_q$  and  $n_p$ 
  - let  $C(n_q)$  and  $C(n_p)$  be the costs of nodes  $n_q$  and  $n_p$
- Expectation:  $C(n_q) = C(n_p)$

- Fraction of excess work:  $X_w(n_q) = \frac{C(n_q) - C(n_p)}{T_q}$  parallel overhead  
total time



# Strong Scaling Analysis for SPMD Codes

## Performance expectation for strong scaling

- work is constant
- execution time decreases linearly with # processors

- Execute code on  $p$  and  $q$  processors; without loss of generality,  $p < q$
- Let  $T_i$  = total execution time on  $i$  processors
- For corresponding nodes  $n_q$  and  $n_p$ 
  - let  $C(n_q)$  and  $C(n_p)$  be the costs of nodes  $n_q$  and  $n_p$
- Expectation:  $qC_q(n_q) = pC_p(n_p)$

- Fraction of excess work:  $X_s(C, n_q) = \frac{qC_q(n_q) - pC_p(n_p)}{qT_q}$  parallel overhead  
total time



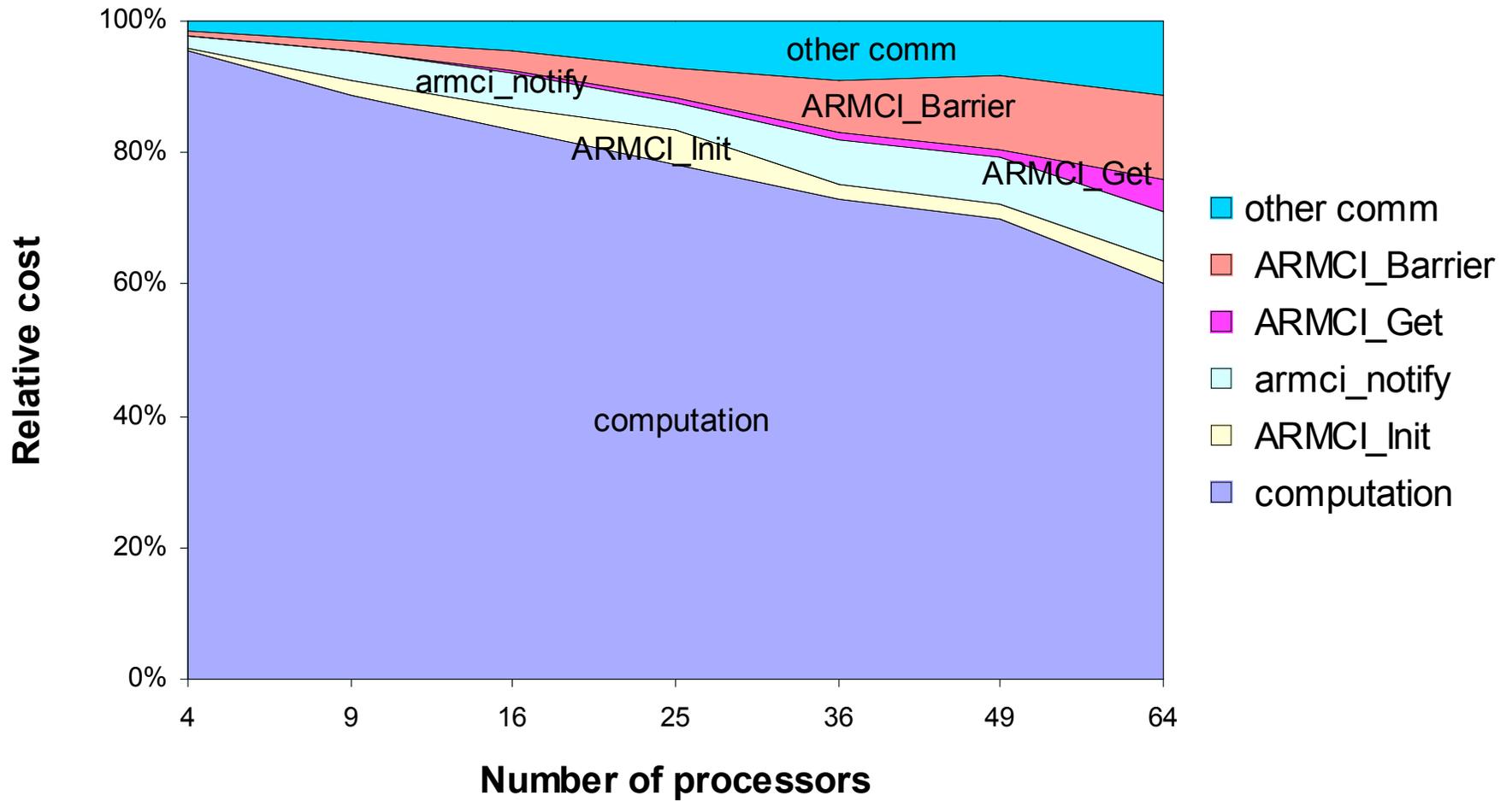
# Scaling Analysis with Expectations

---

- Excess work metrics are intuitive
  - = 0 ideal scaling
  - > 0 suboptimal scaling
- Using excess work metrics
  - $X(I,n) \approx X(E,n)$ : scaling loss due to computation in  $n$
  - $X(I,n) \gg X(E,n)$ : scaling loss due  $n$ 's callees
  - using multiple views
    - losses associated with few calling contexts  $\Rightarrow$  CCT view suffices
    - losses spread across many contexts  $\Rightarrow$  use callers view

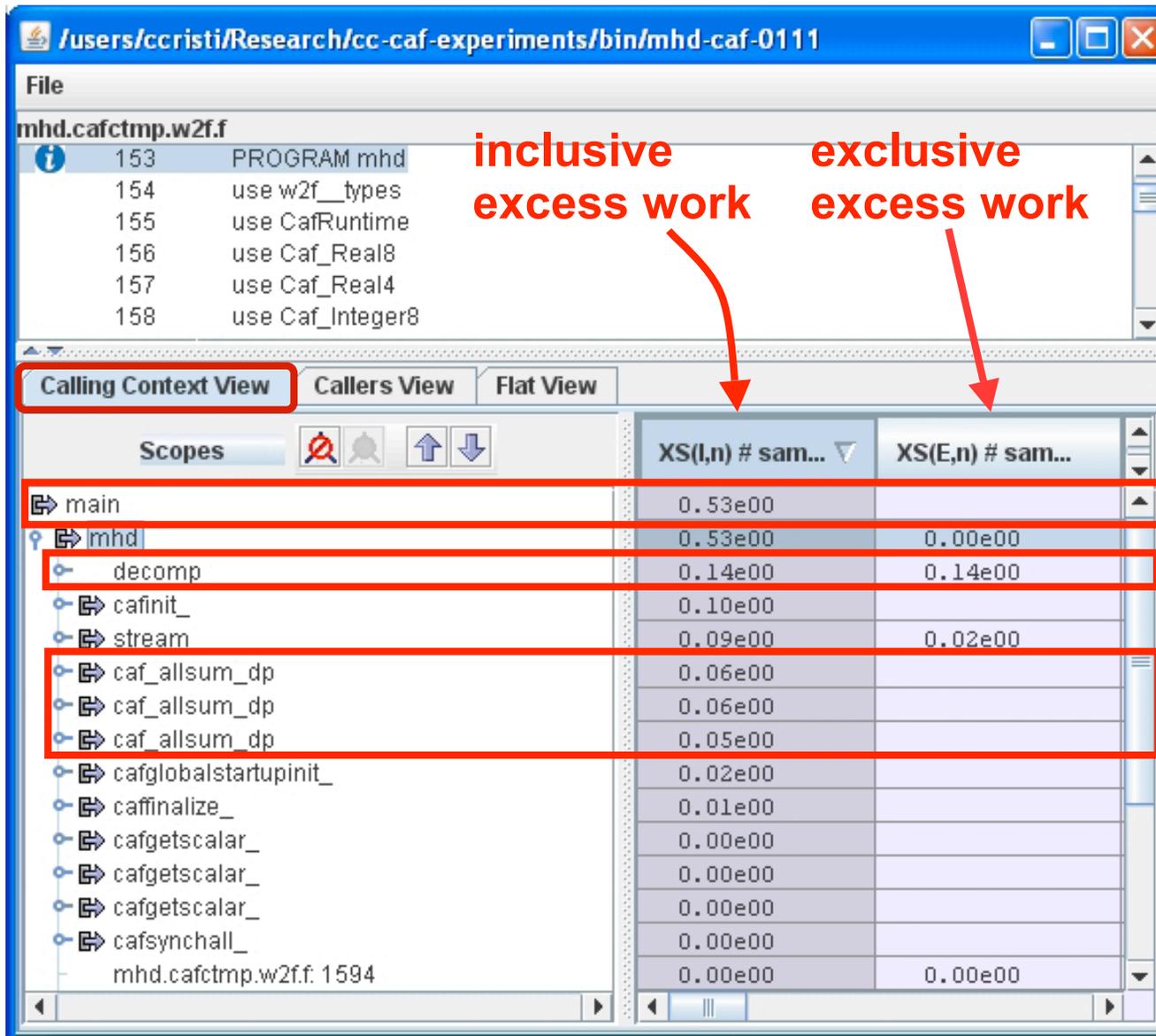


# LBMHD size $1024^2$





# Strong Scaling Analysis of LBMHD



**inclusive  
excess work**

**exclusive  
excess work**

**53% excess work  
= 47% efficiency**

**14% scalability  
loss due to  
computation**

**17% scalability  
loss due to  
barrier-based  
reductions**



# LANL's Parallel Ocean Program (POP)

```
diagnostics.f90
2318
2319   do k=1,km
2320
2321     !*** zonal advection
2322
2323     imax = maxloc(cfl_advuk_block(:,k))
2324     cfl_temp = cfl_advuk_block(imax(1),k)
2325     cfl_advuk(k) = global_maxval(cfl_temp)
2326
2327     cfladd_temp = 0
2328     if (cfl_temp == cfl_advuk(k)) then
2329       cfladd_temp(1:2) = cfladd_advuk_block(:,imax(1),k)
2330     endif
2331     cfladd_advuk(1,k) = global_maxval(cfladd_temp(1))
2332     cfladd_advuk(2,k) = global_maxval(cfladd_temp(2))
2333
```

Calling Context View   **Callers View**   Flat View

Scopes

- main
- pop
- MPID\_RecvComplete
- PMPI\_Waitall
- PMPI\_Sendrecv
  - intra\_Allreduce
  - MPI\_Allreduce
    - mpi\_allreduce\_
      - global\_reductions\_mp\_global\_maxval\_sc
      - diagnostics\_mp\_cfl\_check\_
      - diagnostics\_mp\_cfl\_check\_
      - global\_reductions\_mp\_global\_sum\_sca

XS(l,n) ...	XS(E,n) ...
0.71e00 ...	
0.71e00 ...	
0.60e00 ...	0.00e00 ...
0.31e00 ...	
0.12e00 ...	
0.01e00 ...	
0.00e00 ...	

successive global reductions on scalars degrade parallel efficiency (7 total)

12% loss in scaling due to scalar reductions

7% in this routine alone



# UPC NAS CG class B (size 75000)

```
cg.c
1305 for( i=(l2npcols-1); i>=0; i-- )
1306 {
1307     k = reduce_threads_g[MYTHREAD][i];
1308     l = reduce_rcv_start_g[MYTHREAD][i]-1;
1309     m = reduce_send_start_g[k][i]-1;
1310
1311     upc_memget( prefetch_buffer, &w1[k].arr[m],
1312               sizeof( double ) * reduce_send_len_g[
1313
1314     upc_notify;
1315
1316     for( j=0; j<reduce_send_len_g[k][i]; j++ )
1317     {
1318         w_ptr[l+j] += prefetch_buffer[j];
1319     }
1320
1321
```

Scopes	XS(I,n)	XS(E,n) ...
user_main	0.63e00 ..	0.00e00 ...
loop at cg.c: 409-469	0.60e00 ...	0.00e00 ...
conj_grad	0.60e00 ...	0.08e00 ...
loop at cg.c: 1206-1440	0.57e00 ...	0.08e00 ...
loop at cg.c: 1306-1334	0.33e00 ...	0.05e00 ...
reduce_sum	0.17e00 ...	0.00e00 ...
_upcr_wait	0.03e00 ...	
reduce_sum	0.02e00 ...	0.00e00 ...
loop at cg.c: 1224-1298	0.02e00 ...	0.02e00 ...
loop at cg.c: 1400-1403	0.00e00 ...	0.00e00 ...
loop at cg.c: 1438-1440	0.00e00 ...	0.00e00 ...
_upcr_notify	0.00e00 ...	0.00e00 ...

**63% excess  
work =  
36% efficiency**

**Remote data  
prefetch**



# UPC NAS CG class B (size 75000)

**loss of efficiency due to barrier-based implementation of sum reduction**

```
1543
1544 double reduce_sum( double rs_a)
1545 {
1546 int rs_i;
1547 int rs_o;
1548
1549 #if (TIMERS_ENABLED == TRUE)
1550 timer_start(TIMER_ALLREDUCE);
1551 #endif
1552 upc_barrier;
1553
1554 rs_o = (nr_row * NUM_PROC_COLS);
1555 for( rs_i=rs_o; rs_i<(rs_o+NUM_PROC_COLS); rs_i++)
1556 {
1557 if( rs_i == MYTHREAD )
1558 {
```

**Calling Context View** | Callers View | Flat View

Scopes

- user\_main
  - loop at cg.c: 409-469
    - conj\_grad
      - loop at cg.c: 1206-1440
        - loop at cg.c: 1306-1334
          - reduce\_sum**
          - \_upcr\_wait
          - reduce\_sum
        - loop at cg.c: 1224-1298
        - loop at cg.c: 1400-1403
        - loop at cg.c: 1438-1440

XS(l,n) ...	XS(E,n) ...
0.63e00 ...	0.00e00 ...
0.60e00 ...	0.00e00 ...
0.60e00 ...	0.08e00 ...
0.57e00 ...	0.08e00 ...
0.33e00 ...	0.05e00 ...
<b>0.17e00 ...</b>	<b>0.00e00 ...</b>
0.03e00 ...	...
0.02e00 ...	0.00e00 ...
0.02e00 ...	0.02e00 ...
0.00e00 ...	0.00e00 ...
0.00e00 ...	0.00e00 ...



# Weak Scaling Analysis of MILC's su3\_rmd

The screenshot shows a performance analysis tool window titled "su3\_rmd". The top part of the window displays a code editor with the following C code:

```
1581 void
1582 wait_gather(msg_tag *mtag)
1583 {
1584     MPI_Status status;
1585     int i;
```

Below the code editor is the "Calling Context View" tab, which displays a table of performance metrics for various scopes. The table has two main columns: "XW(I,n) # ..." and "XW(E,n) ...". The "main" and "wait\_gather" rows are highlighted with red boxes.

Scopes	XW(I,n) # ...	XW(E,n) ...
main	0.33e00 ...	0.00e00 ...
update	0.31e00 ...	0.00e00 ...
path_product	0.24e00 ...	-0.01e00 ...
wait_gather	0.22e00 ...	
path_product	0.05e00 ...	
path_product	0.04e00 ...	
u_shift_hw_fermion	0.03e00 ...	
path_product	0.02e00 ...	
path_product	0.02e00 ...	
path_product	0.02e00 ...	
path_product	0.01e00 ...	
path_product	0.01e00 ...	
dslash_fn_field_special	0.01e00 ...	
dslash_fn_field_special	0.00e00 ...	
dslash_fn_field_special	0.00e00 ...	
dslash_fn_field_special	0.00e00 ...	



# Scalability Analysis Using Expectations

---

- Broadly applicable
  - independent of programming model
  - independent of bottleneck cause
  - applicable to a wide range of applications and architectures
- Easy to understand and use
  - fraction of excess work is intuitive and relevant metric
  - attribution to calling context enables precise diagnosis of bottlenecks
  - provides quantitative feedback
- Perfectly suited to petascale systems
  - call stack sampling is efficient enough for production use
  - uses only local performance information
  - data volume is modest and scales linearly
- Drawback
  - pinpoints bottleneck, but provides no intuition into cause



# Outline

---

- Sampling based measurement
- Binary analysis
- User interface
- Scalability analysis
-  Components
  - ours
  - our desires
- Related modeling activities



# Components to Share

---

- libmonitor - infrastructure for augmenting program with monitoring
  - what
    - monitors program launch thread creation/termination, fork/exec, exit
  - how
    - preloaded library for dynamically linked executables
    - static library for statically-linked executables
- hpcviewer user interface
  - three views: calling context, caller's view, flat view
  - scalability analysis
- bloop binary analyzer
  - identify loops, inlined code
- OpenAnalysis - representation-independent program analysis tools
  - call graph and control-flow graph construction
  - dataflow analysis



# Component Needs

---

- Metadata collection
- Standard OS interface for sampling-based measurement
- Ubiquitous stack unwinder for fully-optimized code
  - instruction cracker
  - engine for recovering frame state info at any point in an execution



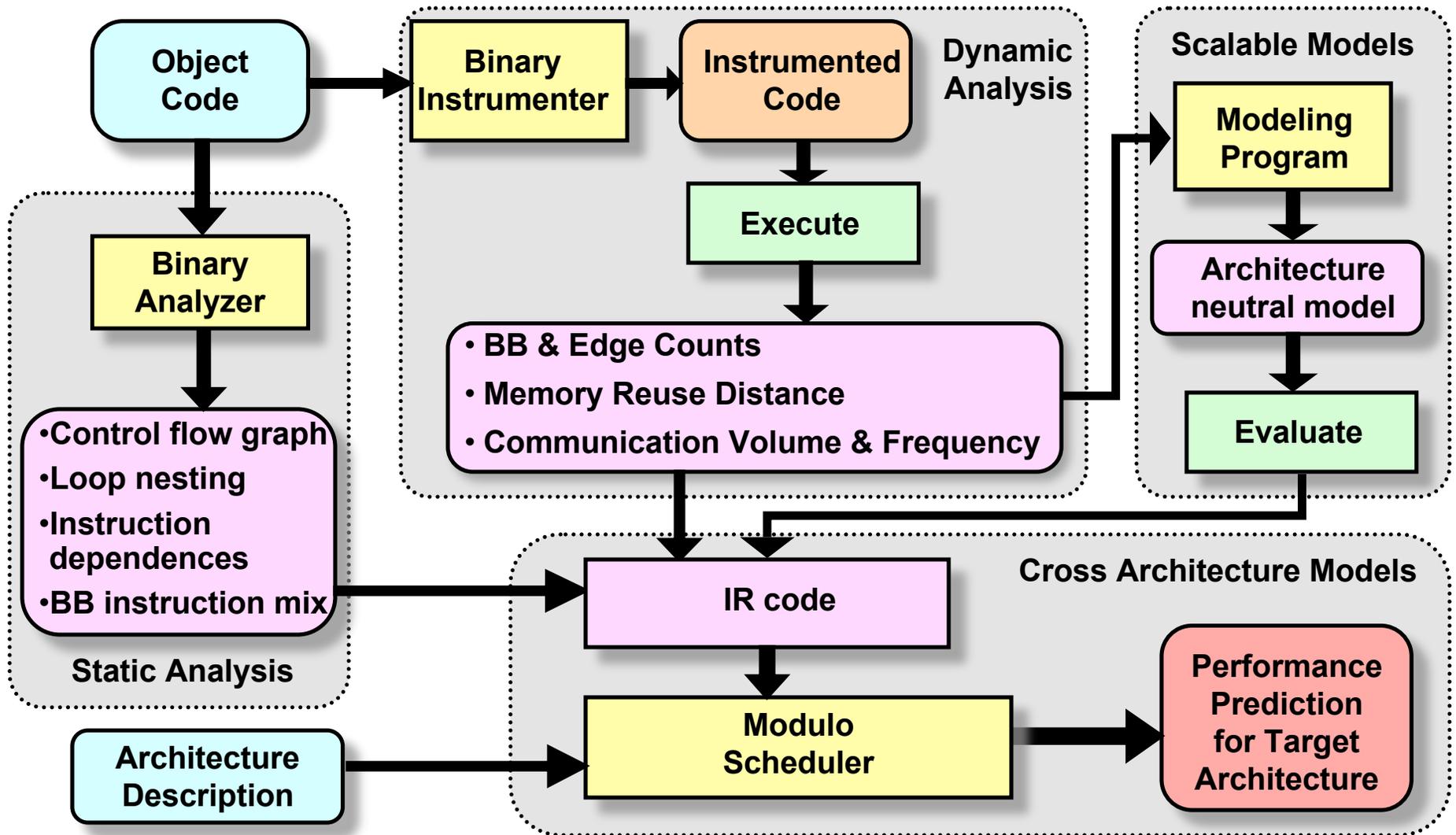
# Outline

---

- Sampling based measurement
- Binary analysis
- User interface
- Scalability analysis
- Components
  - ours
  - our desires
- 👉 **Related modeling activities**



# Analysis and Modeling of Node Performance





# Capabilities of Modeling Toolkit

---

## Loop level attribution of metrics

- Attribute execution costs to underlying causes
  - data dependencies that serialize operations
  - insufficient CPU resources
  - memory delays (latency and bandwidth)
- Explain patterns of data reuse
  - pinpoint opportunities for enhancing temporal reuse
  - pinpoint low spatial reuse
- Automatic “what if” scenarios
  - infinite number of CPU resources
  - no register or memory dependencies
  - no memory delays