Paradyn Parallel Performance Tools

Barton P. Miller, Jeff Hollingsworth
bart@cs.wisc.edu, hollings@cs.umd.edu

Mehmet Altinel (UMD)
Drew Bernat
Bryan Buck (UMD)
Trey Cain
Chris Chambreau
Mihai Christodorescu
Kyong Dong (UMD)
Nick Rasmussen
Philip Roth
Brandon Schendel
Ari Tamches
Mustafa Tikir
Roland Wismüller (TUM)
Brian Wylie
Zhichen Xu
Victor Zandy
Wei Zhang

Some History and Motivation

Experience with IPS-2 tools project:

- Trace-based tool running on workstations, SMP (Sequent Symmetry), Cray Y-MP.
- Commercial Success: In Sun SPARCWorks, Informix OnView, NSF Supercomputer Centers.
- Many real scientific and database/transaction users.

More Motivation and History

A 1992 Design Meeting at Intel:

- Goal was to identify hardware support for profiling and debugging for the Touchstone Sigma (Paragon) machine.
- We estimated the cost of tracing, using IPS-2 experience, and extrapolated to the new machine.
- We predicted 2 MB/sec/node $\rightarrow$ 2 GB/sec of trace data for 1000 node machine.

We went home to Madison to re-think this.

The Challenges

Scalability:

- Large Programs
- 100's or 1000's of nodes
- Long runs (hours or days)

Automate Tuning Process:

- Simplify the task of programmer
- Deal with increasing complexity.

Support Heterogeneity:

- COWs, SMPs, MPPs.
- UNIX, NT, Linux

Extensible:

- Incorporate new sources of performance data.
- Include new visualizations.

Searching for Bottlenecks

1. Start with coarse-grain view of whole program performance
2. When you see a problem, collect more information to refine this problem.
3. Repeat step #2 until you have a precise enough cause.
4. Collect information to try to refine to particular hosts, processes, modules, functions, files, etc.
5. Repeat step #4 until you have a precise enough location.

This type of iteration can take a user many runs of a program to reach a useful conclusion.

Approach the Problem Differently: Do Everything Dynamically

Paradyn allows the programmer to do this on-the-fly in a single execution.

The Major Technologies

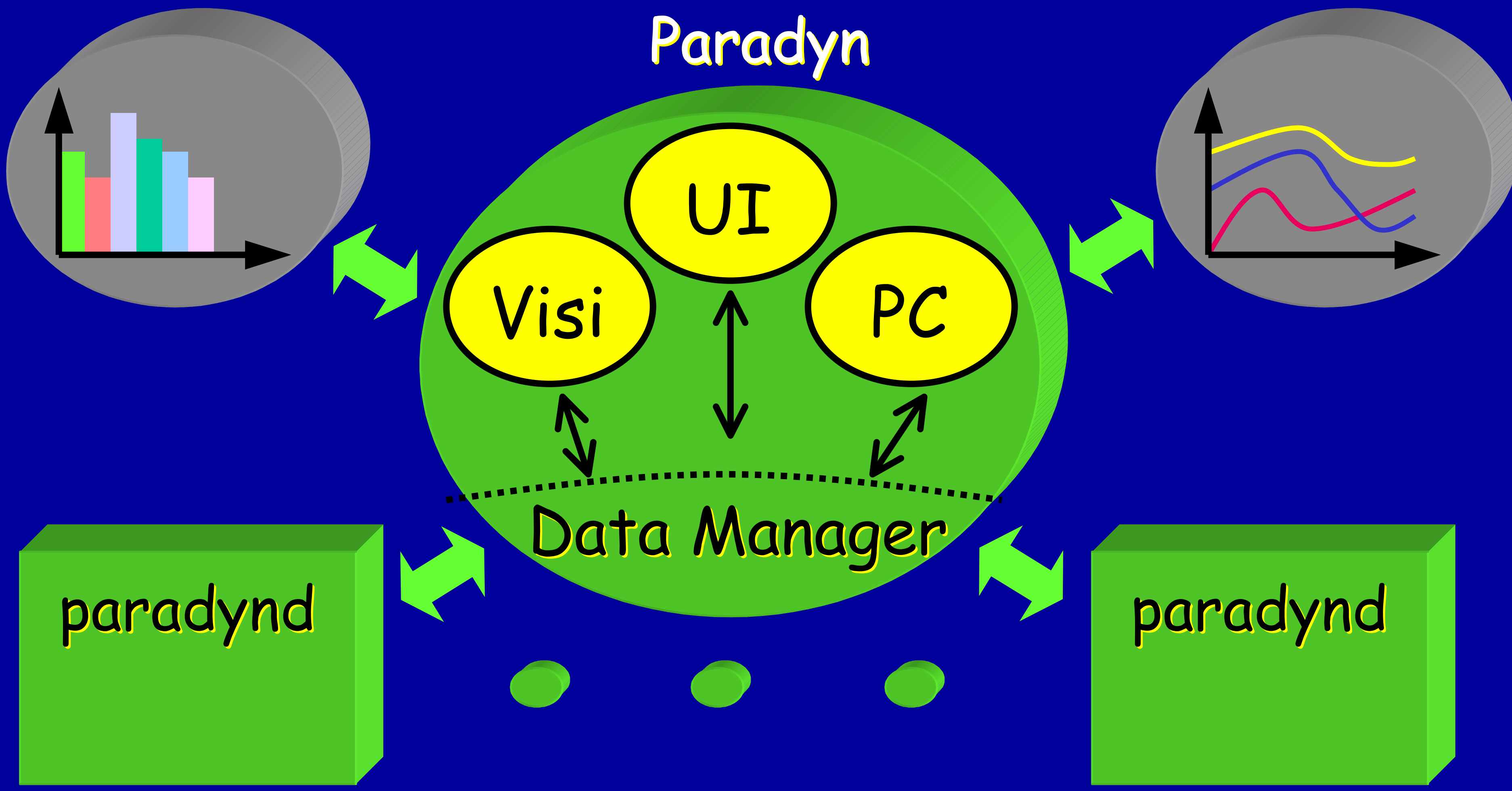
Dynamic Instrumentation

- On-the-fly: Insert, remove, and change instrumentation in the application program while it is running.

Automating the Search for Bottlenecks

- The Performance Consultant: identify bottlenecks and automate control the the Dynamic Instrumentation.

Paradyn Architecture



Dynamic Instrumentation

- ❑ Does not require recompiling or relinking
  - Saves time: compile and link times are significant in real systems.
  - Can profile without the source code (e.g., proprietary libraries).
  - Can profile without linking (relinking is not always possible.
- ❑ Instrument optimized code.

Dynamic Instrumentation (con'd)

- Only instrument what you need, when you need
  - No hidden cost of latent instrumentation.
  - Enables "one pass" performance sessions.

- Can monitor running programs (such as database servers)
  - Production systems.
  - Embedded systems.
  - Systems with complex start-up procedures.

Dynamic Instrumentation (con'd)

- Anything in the application's address space can become a performance measure
  - Application metrics: transactions/second, commits/second.
  - OS metrics: page faults, context switches, disk I/O's.
  - Hardware metrics: cycle & instruction counters, miss rates, network statistics.
  - User-level protocol metrics: Blizzard messages, cache activity.

Dynamic Instrumentation (con'd)

- New metrics defined through our Metric Description Language (MDL)
  - Neither performance tool nor application need be modified.
  - Define metrics once for each environment.
- Can dynamically monitor and control instrumentation overhead.
  - Allows programmer to *monitor* intrusiveness
  - Allows programmer to *control* intrusiveness.
  - Allows Perf Consultant to work efficiently.

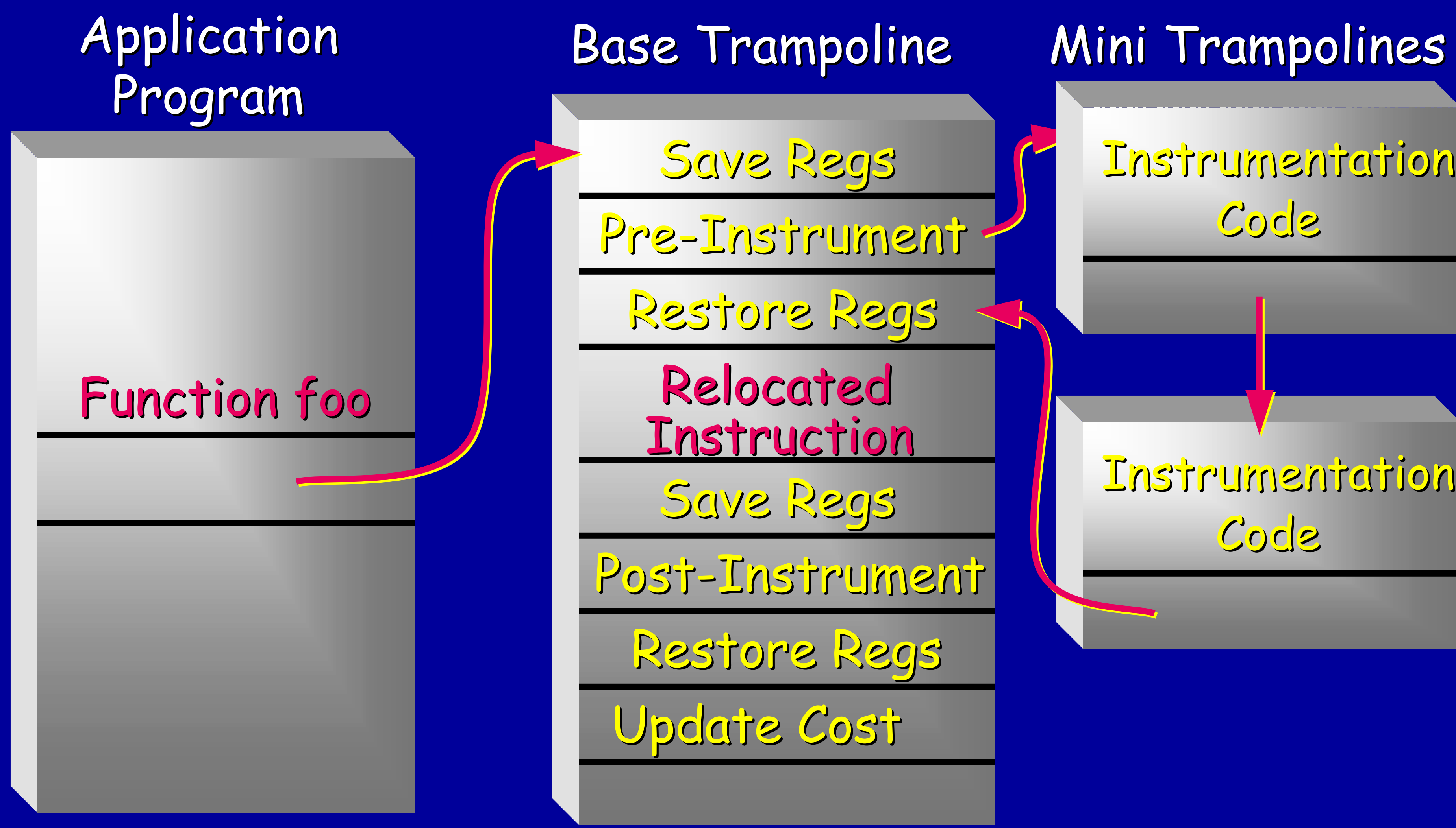
Dynamic Instrumentation Challenges

- Finding instrumentation points (function entry, exit, call site).
  - Procedure exit is often the toughest.
- Finding space for jump to trampoline
  - Long jumps (2-5 words or 5 bytes)
  - Short code sequences
  - Small functions
  - One byte instructions.

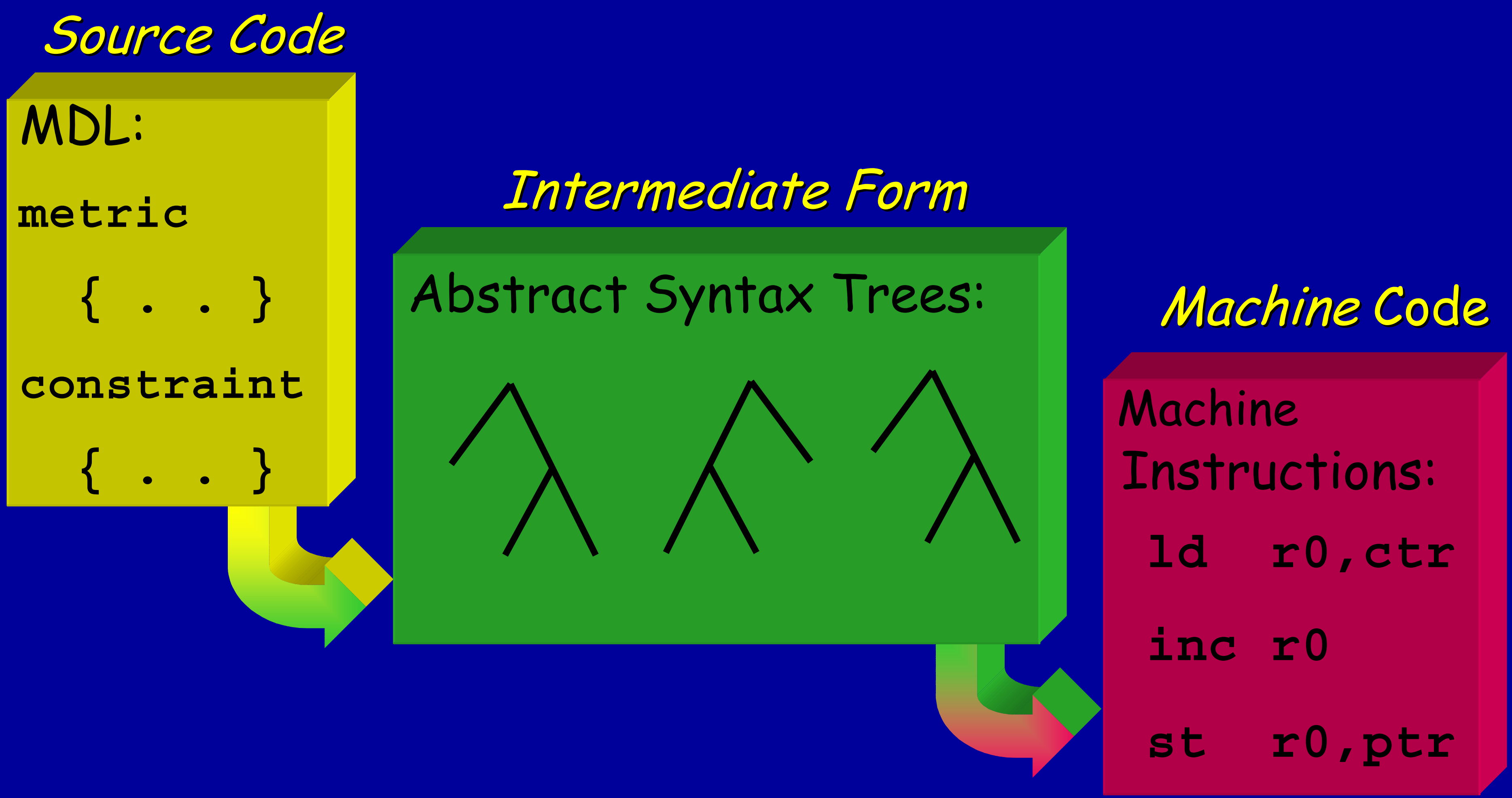
Dynamic Instrumentation Challenges (con'd)

- Compiler optimizations (we instrument optimized code)
  - No explicit stack frame (leaf functions)
  - Tight instrumentation points.
  - Data in code space (e.g., jump tables)
  - De-optimize code on the fly (e.g., tail calls)
- Threaded code
  - Multiple threads executing the same instrumentation code; very tricky!

Patching in Instrumentation

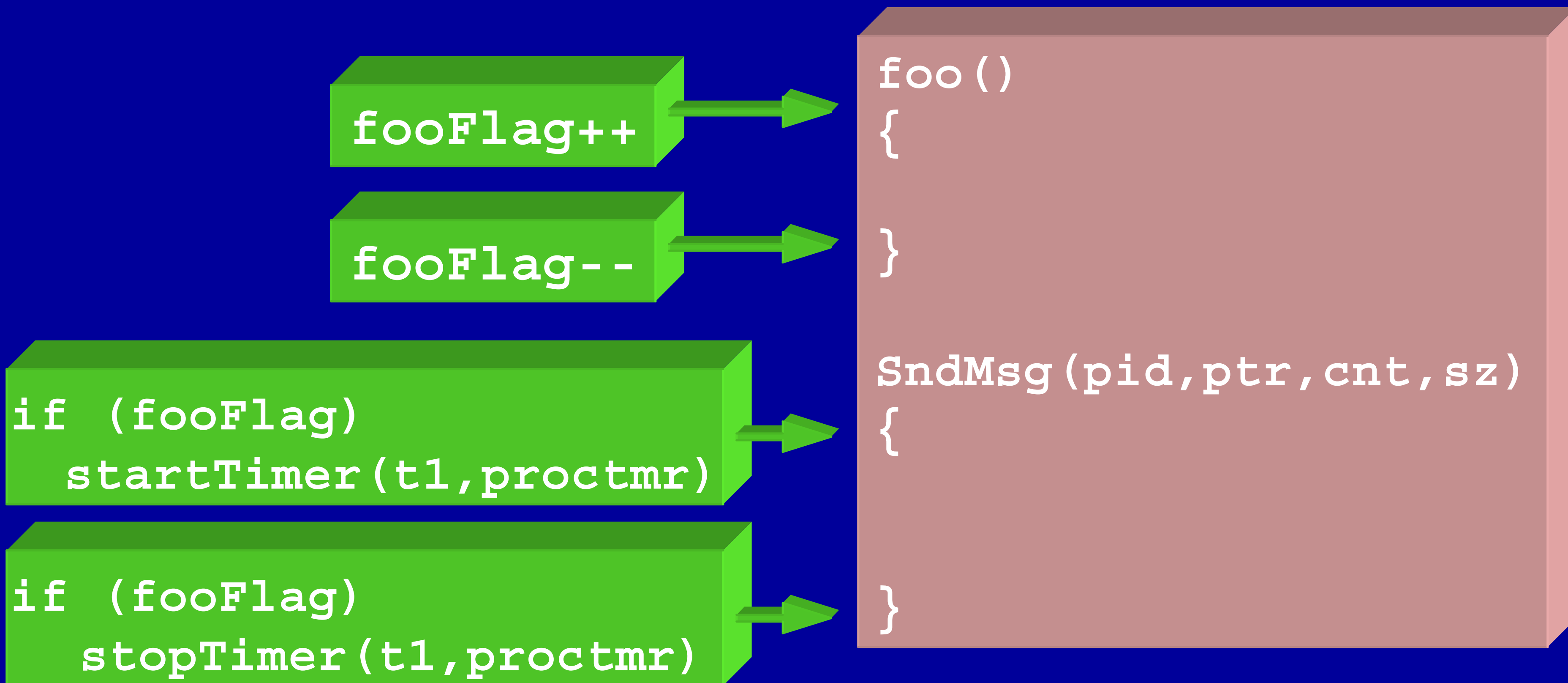


Compiling for Dynamic Instrumentation



Basic Instrumentation Operations

- Points: places to insert instrumentation
- Primitives: code that gets inserted



Decision Support ("Performance Consultant")

Answer three questions:

- Why is the program running slowly?
- Where in the program is this occurring?
- When does this problem occur?

We create a regular structure for the causes of bottlenecks.

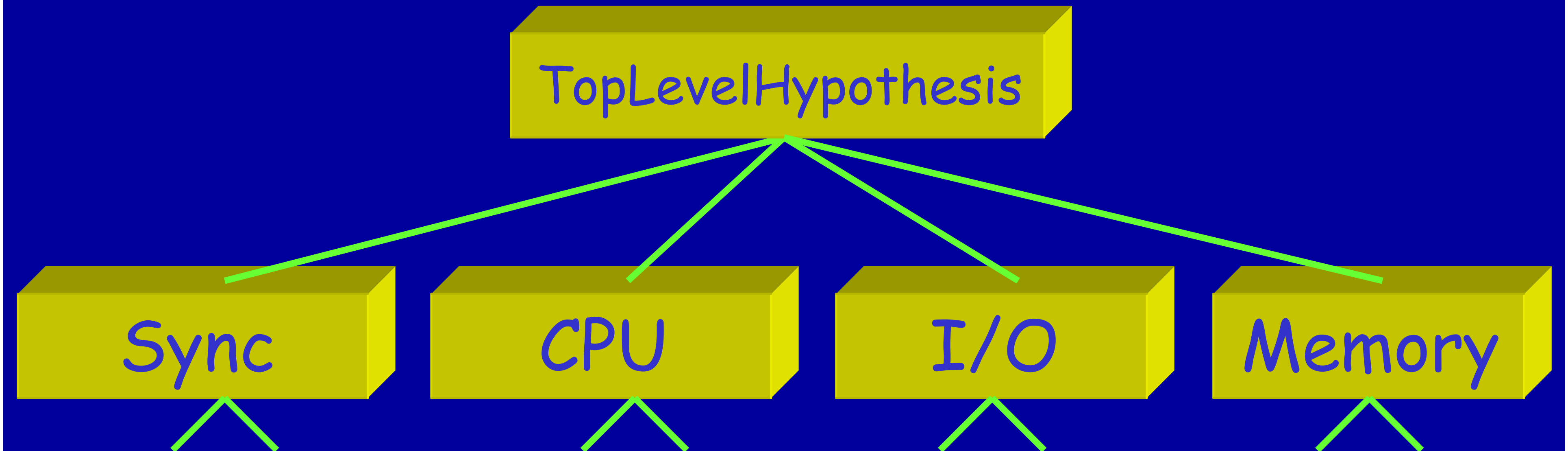
This makes it possible to automate the search for bottlenecks.

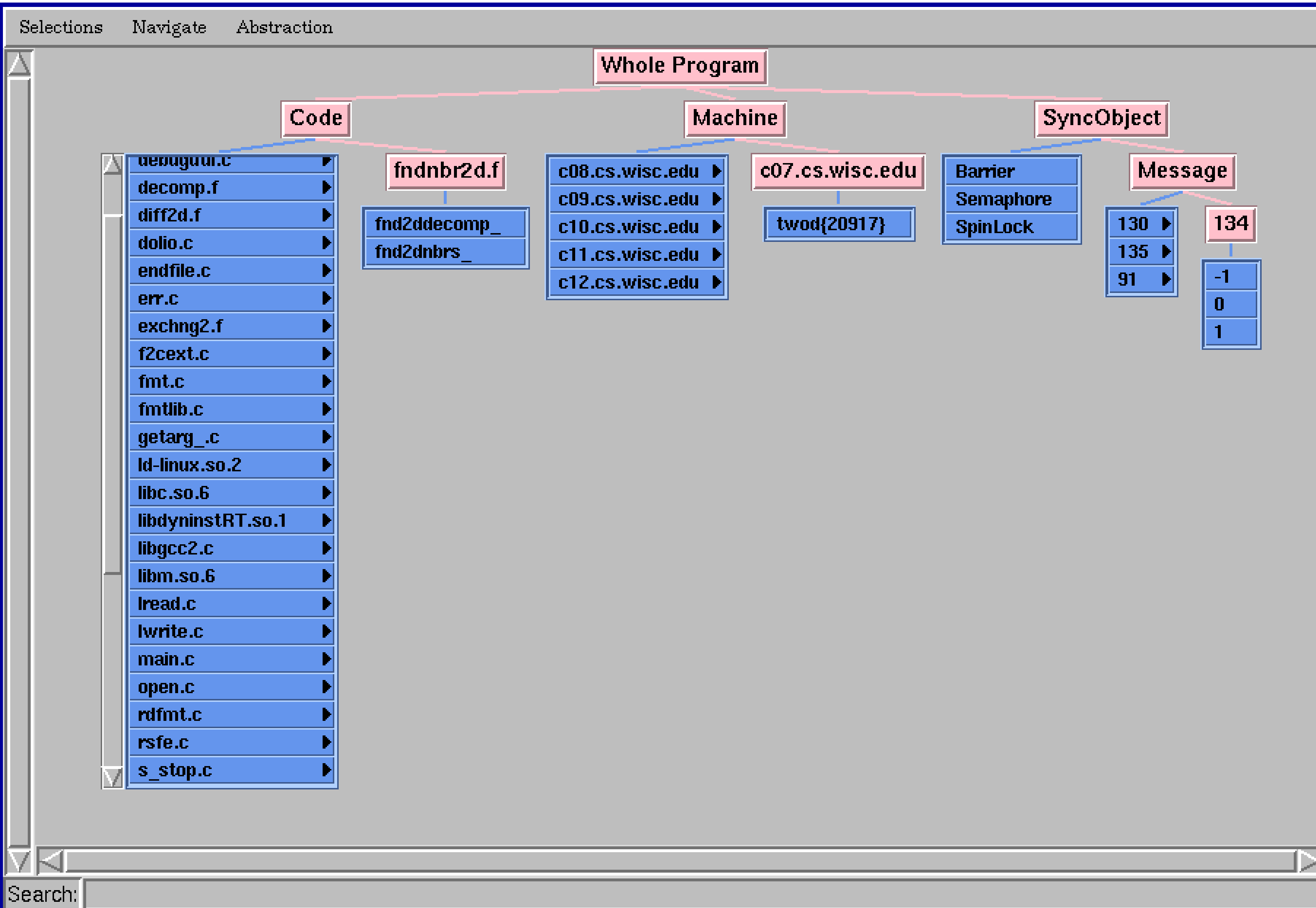
The "Why" Axis: A Hierarchy of Bottlenecks

- Potential bottlenecks are represented as hypotheses.
- Evaluating hypotheses triggers dynamic instrumentation.
- Bottlenecks are based on user-set thresholds:

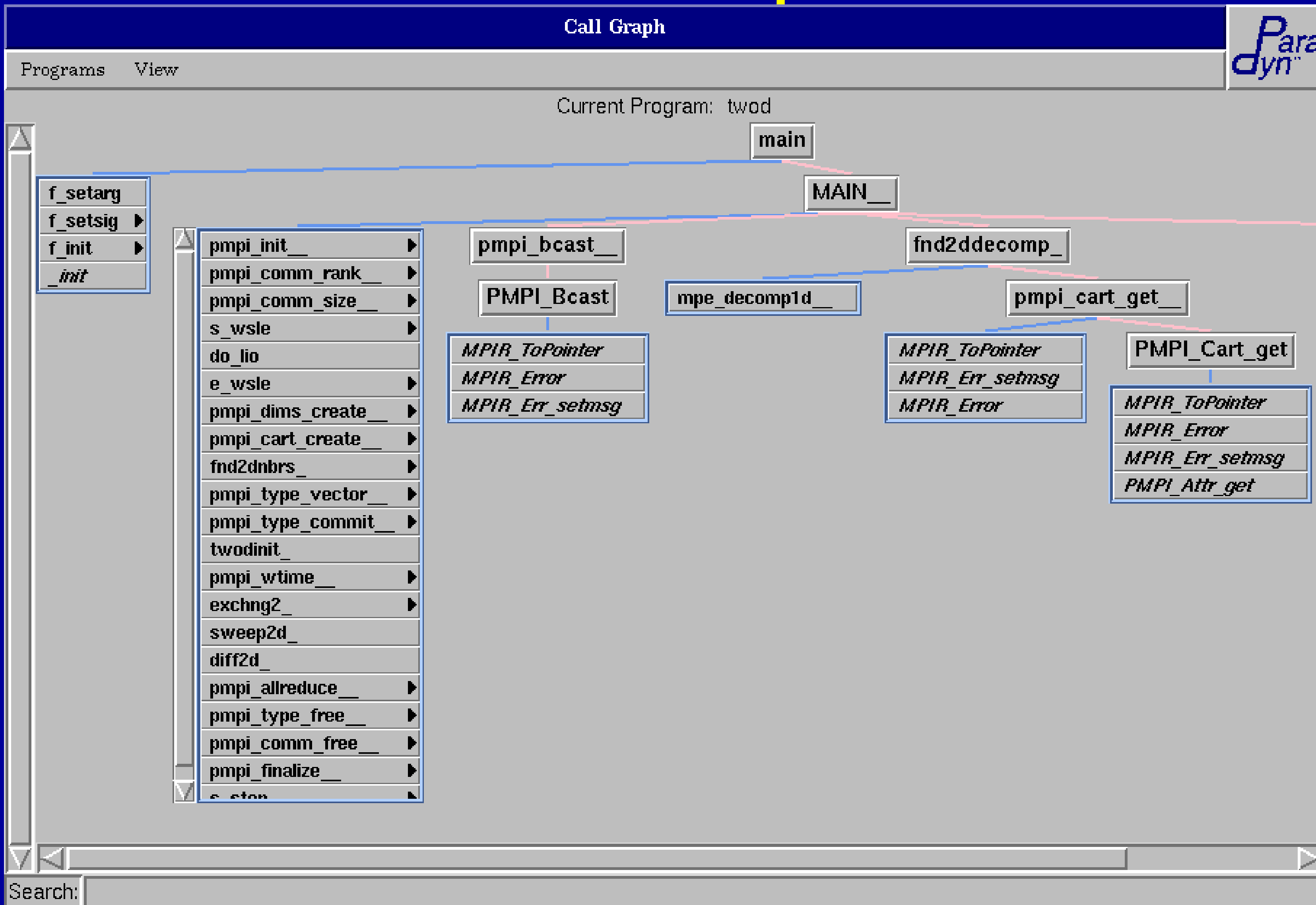
  Total sync blocking time < 25% of exec time

The "Why" Axis: A Hierarchy of Bottlenecks

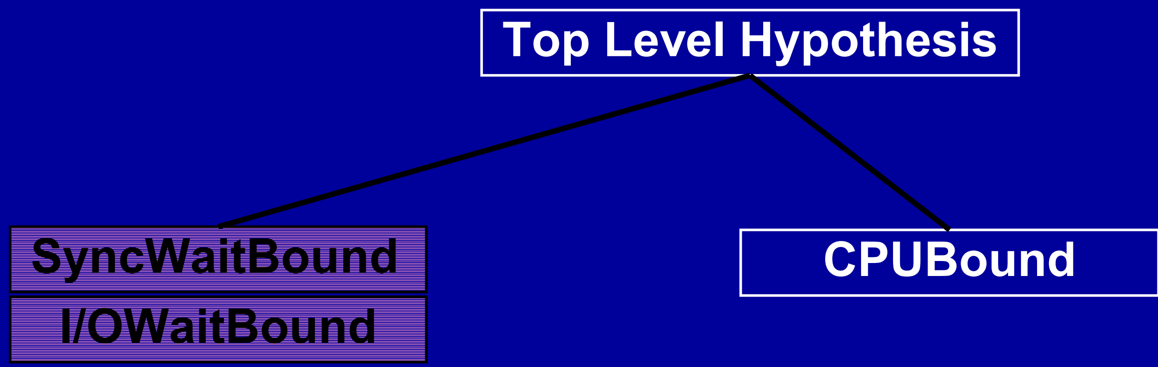



Selections
Navigate
Abstraction
Whole Program
Code
Machine
SyncObject
decomp.f
diff2d.f
dolio.c
endfile.c
err.c
exchng2.f
f2cext.c
fmt.c
fmtlib.c
getarg_.c
ld-linux.so.2
libc.so.6
libdyninstRT.so.1
libgcc2.c
libm.so.6
lread.c
lwrite.c
main.c
open.c
rdfmt.c
rsfe.c
s_stop.c
fndnbr2d.f
fnd2ddecomp_
fnd2dnbrs_
c08.cs.wisc.edu
c09.cs.wisc.edu
c10.cs.wisc.edu
c11.cs.wisc.edu
c12.cs.wisc.edu
c07.cs.wisc.edu
twod{20917}
Barrier
Semaphore
SpinLock
Message
130
135
91
134
-1
0
1
Search:

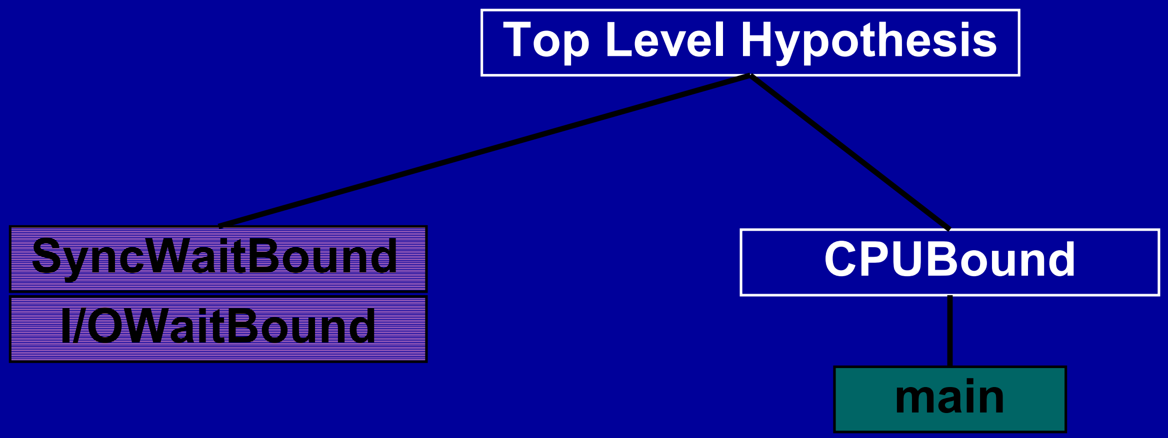
Call Graph


Call Graph
Programs
View
Current Program: twod
main
MAIN__
f_setarg
f_setsig
f_init
_init
pmpi_init__
pmpi_comm_rank__
pmpi_comm_size__
s_wsle
do_lio
e_wsle
pmpi_dims_create__
pmpi_cart_create__
fnd2dnbrs_
pmpi_type_vector__
pmpi_type_commit__
twodinit_
pmpi_wtime__
exchng2_
sweep2d_
diff2d_
pmpi_allreduce__
pmpi_type_free__
pmpi_comm_free__
pmpi_finalize__
pmpi_bcast__
PMPI_Bcast
MPIR_ToPointer
MPIR_Error
MPIR_Err_setmsg
mpe_decomp1d__
fnd2ddecomp_
pmpi_cart_get__
MPIR_ToPointer
MPIR_Err_setmsg
MPIR_Error
PMPI_Cart_get
MPIR_ToPointer
MPIR_Error
MPIR_Err_setmsg
PMPI_Attr_get
Search:

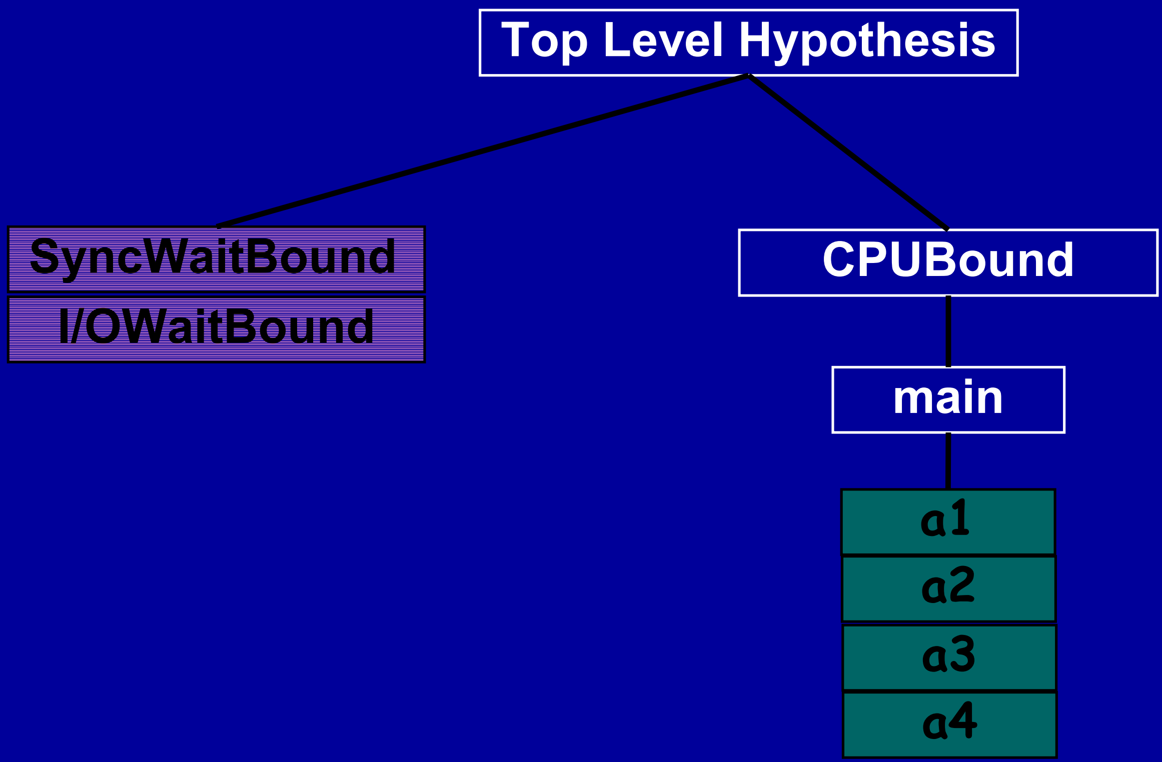
Call Graph Based PC Example



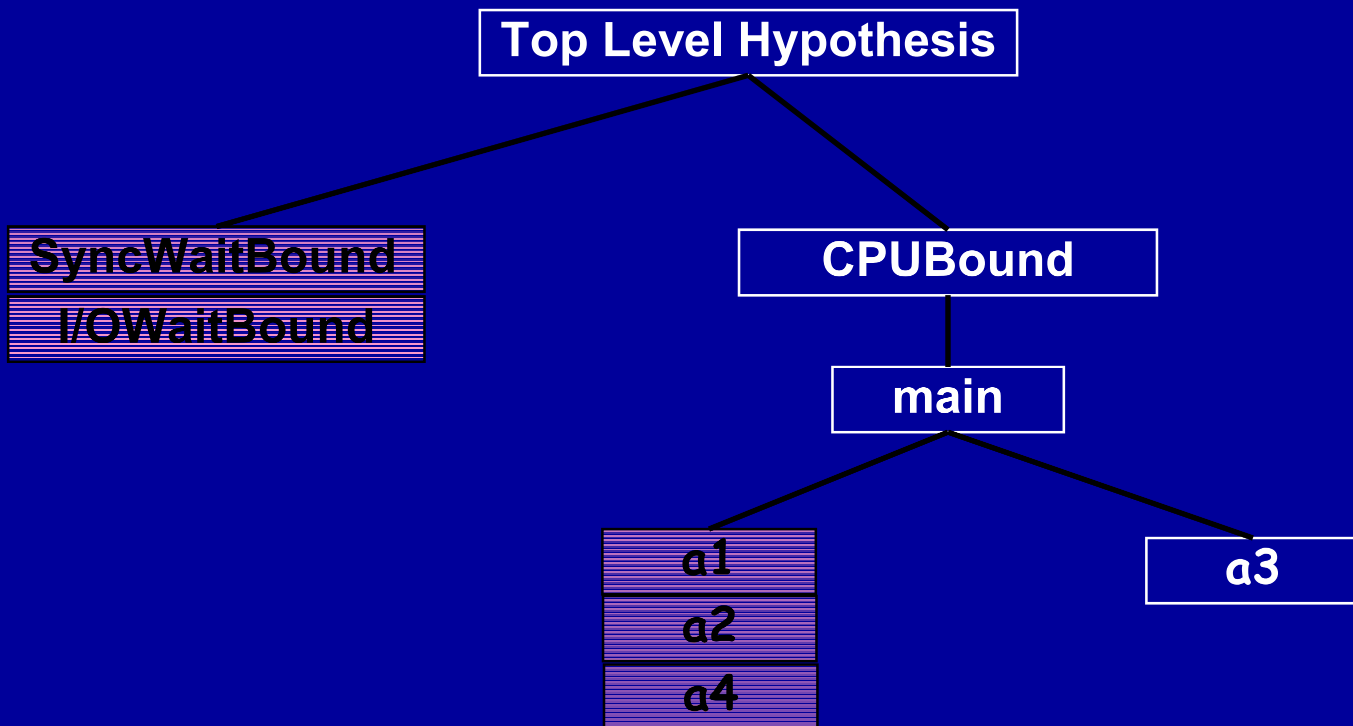
Call Graph Based PC Example



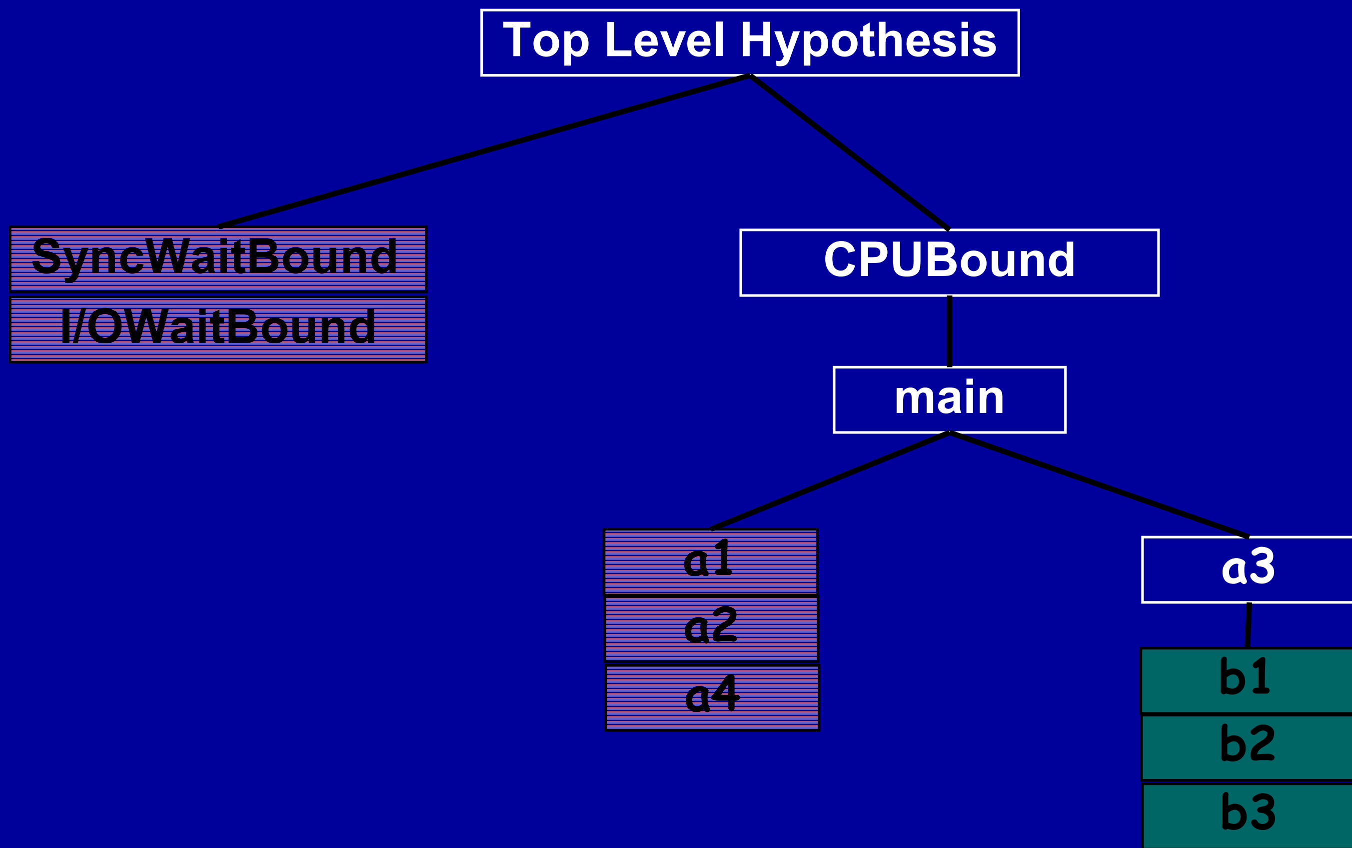
Call Graph Based PC Example



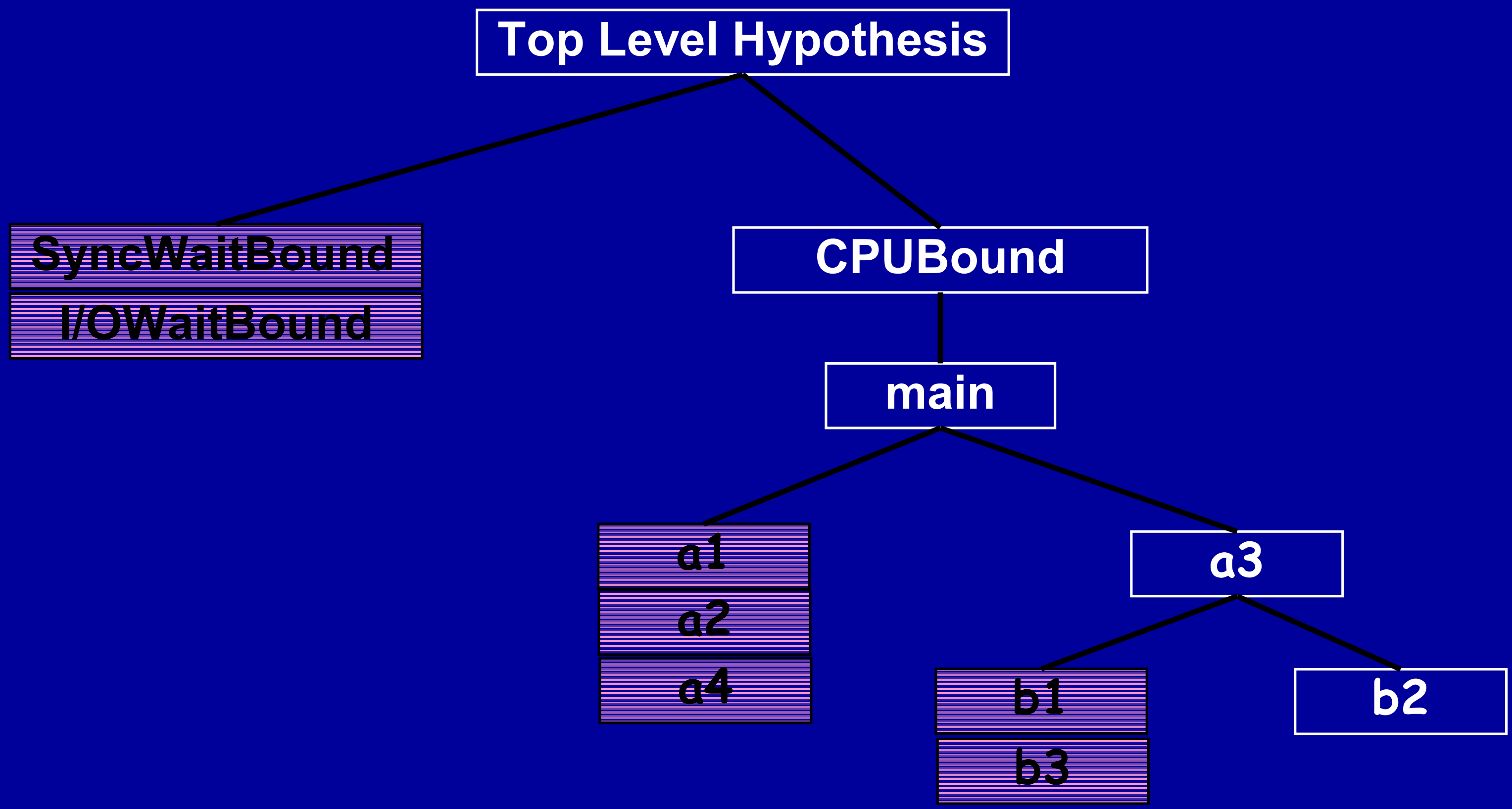
Call Graph Based PC Example



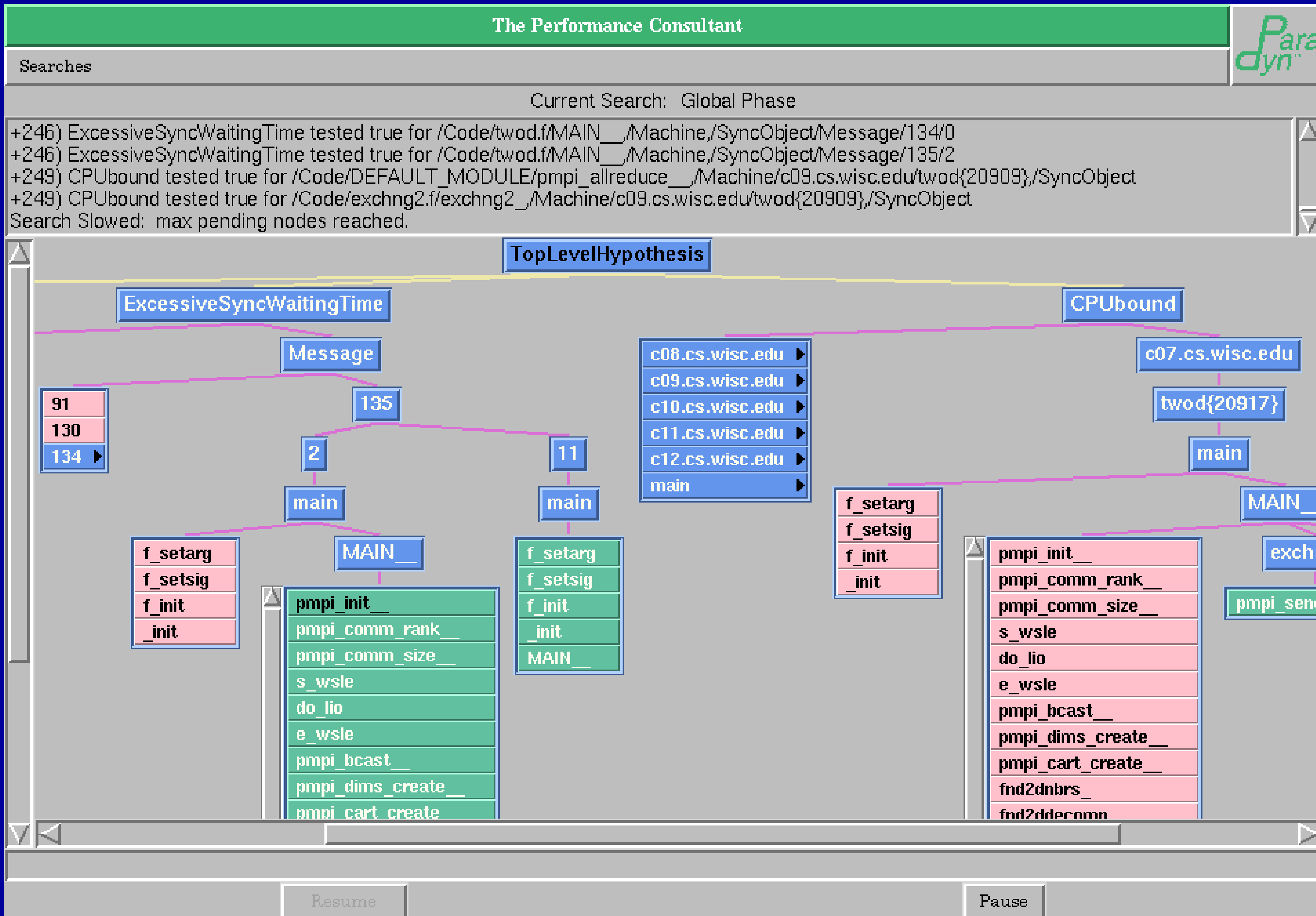
Call Graph Based PC Example



Call Graph Based PC Example



Performance Consultant (SHG)



Call Graph Construction

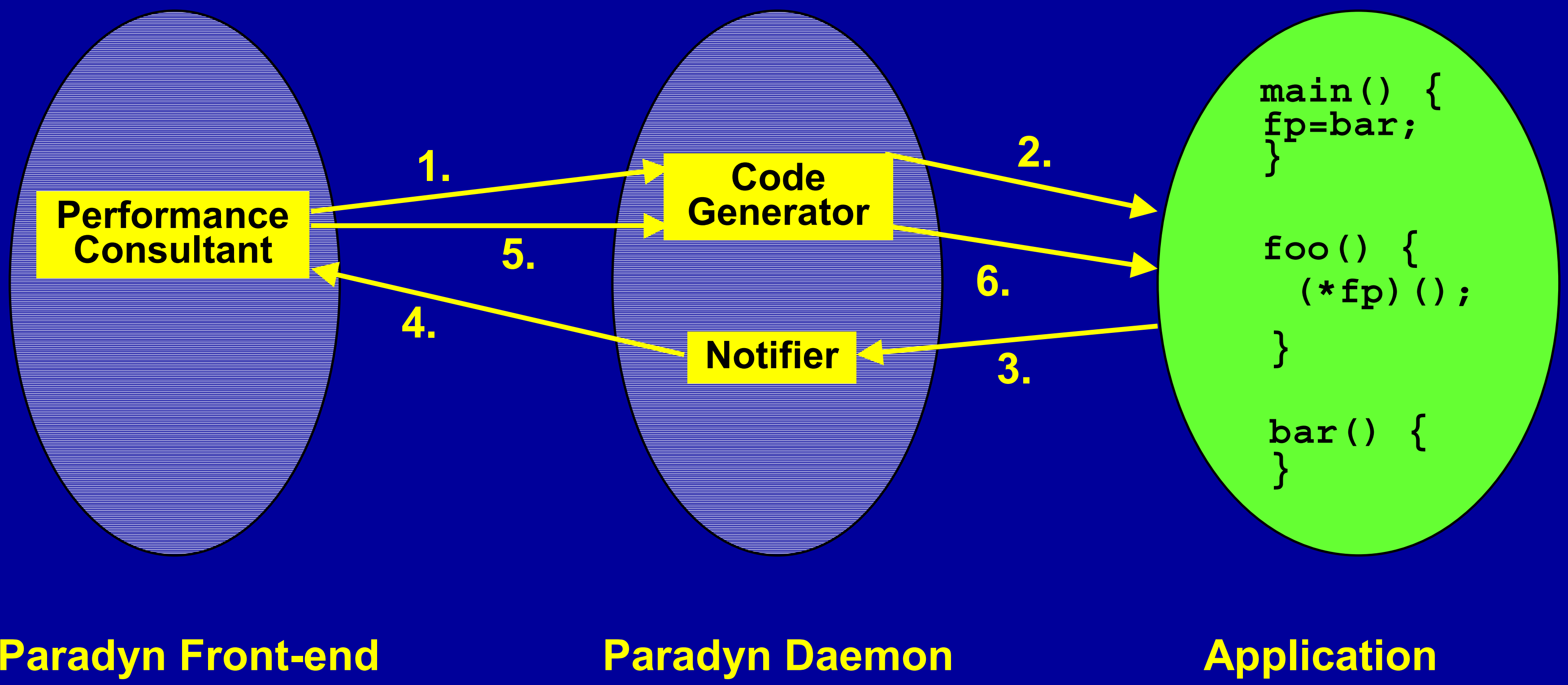
- Problem: targets of calls using function pointers and virtual functions are not statically determinable.
- Unknown callees in static call graph may cause blind spots in new PC search
- We resolve dynamic callee addresses at run time
- Strategy:
  - Build static call graph at program start
  - Fill in dynamic call graph on demand.

Dynamic Call Sites

- Characterized by keeping the address of a callee in a register or memory location
- New type of instrumentation necessary to determine callee
- Examples:

Instruction Set	Call Instruction
MIPS	`jalr $t9`
X86	`call [%edi]`

Call Site Instrumentation: Chain of Events



Controlling Instrumentation Cost

"What is the overhead of instrumentation?"

translates to:

"How many hypotheses do we evaluate at once?"

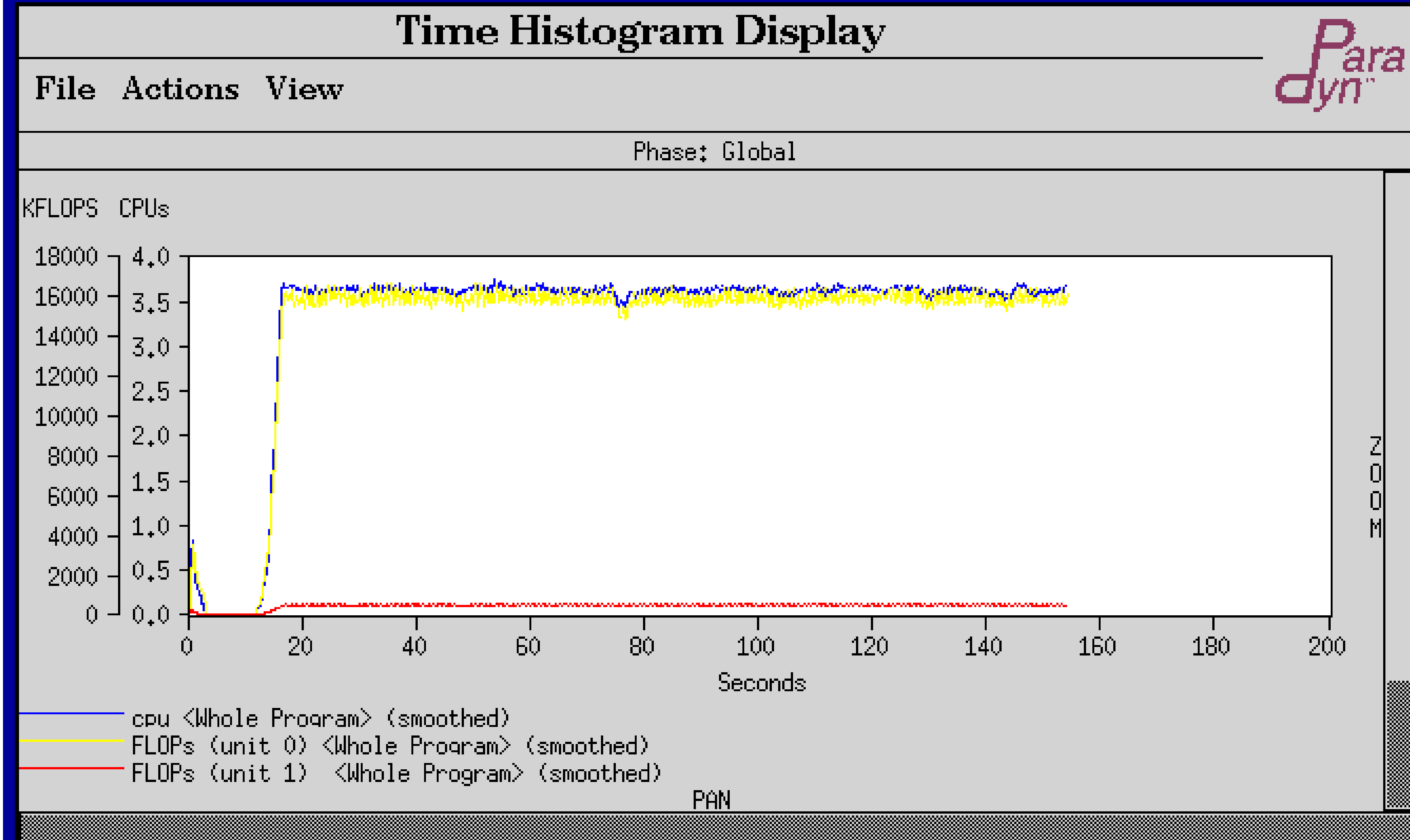
Predicted Cost:

- Known primitive cost
- Estimate frequency
- User-defined threshold

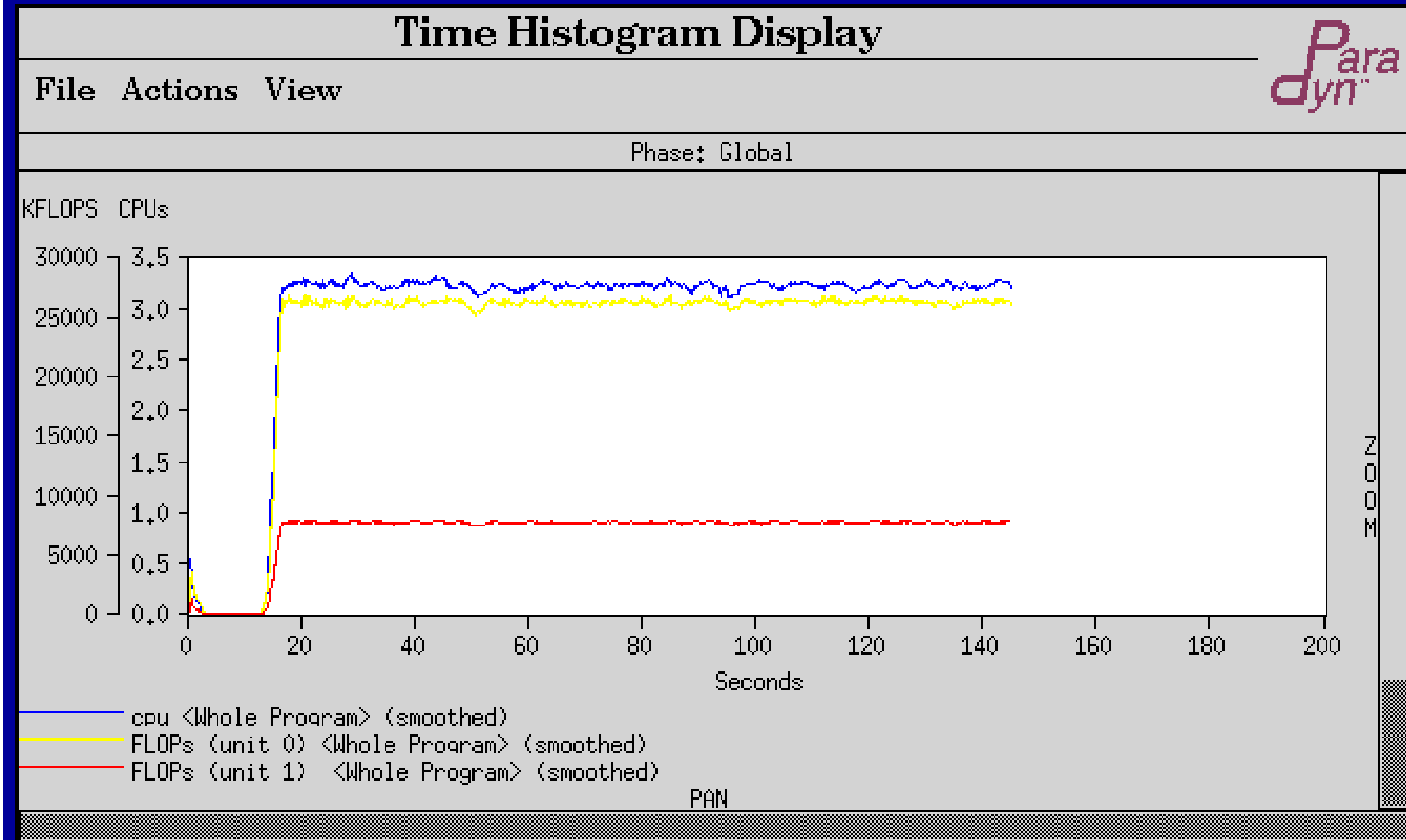
Observed Cost:

- Calculates actual cost
- Meta-instrumentation
- Reports to Performance Consultant

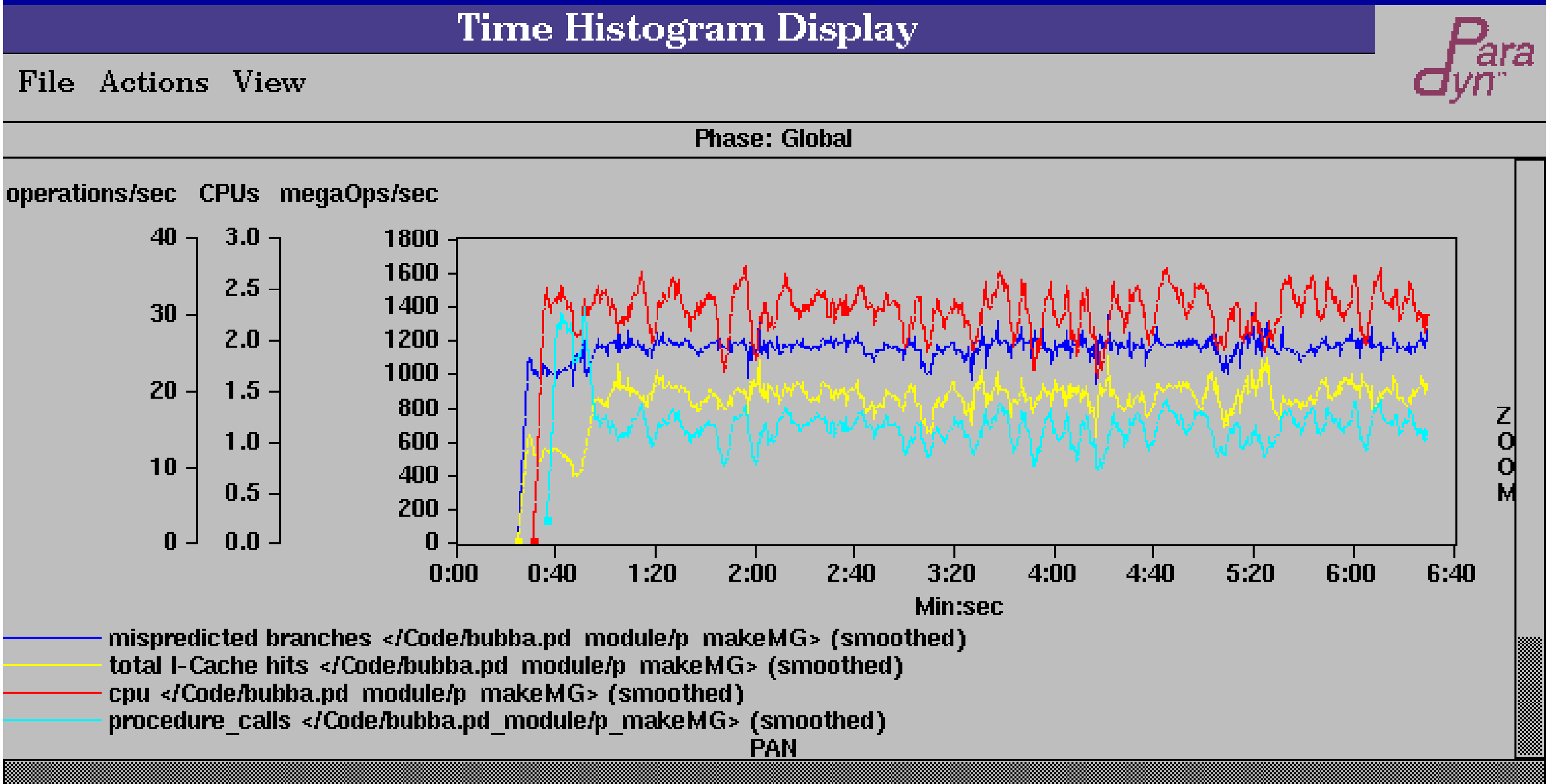
SP2 Hardware Perf Counters (gcc2.6.3)



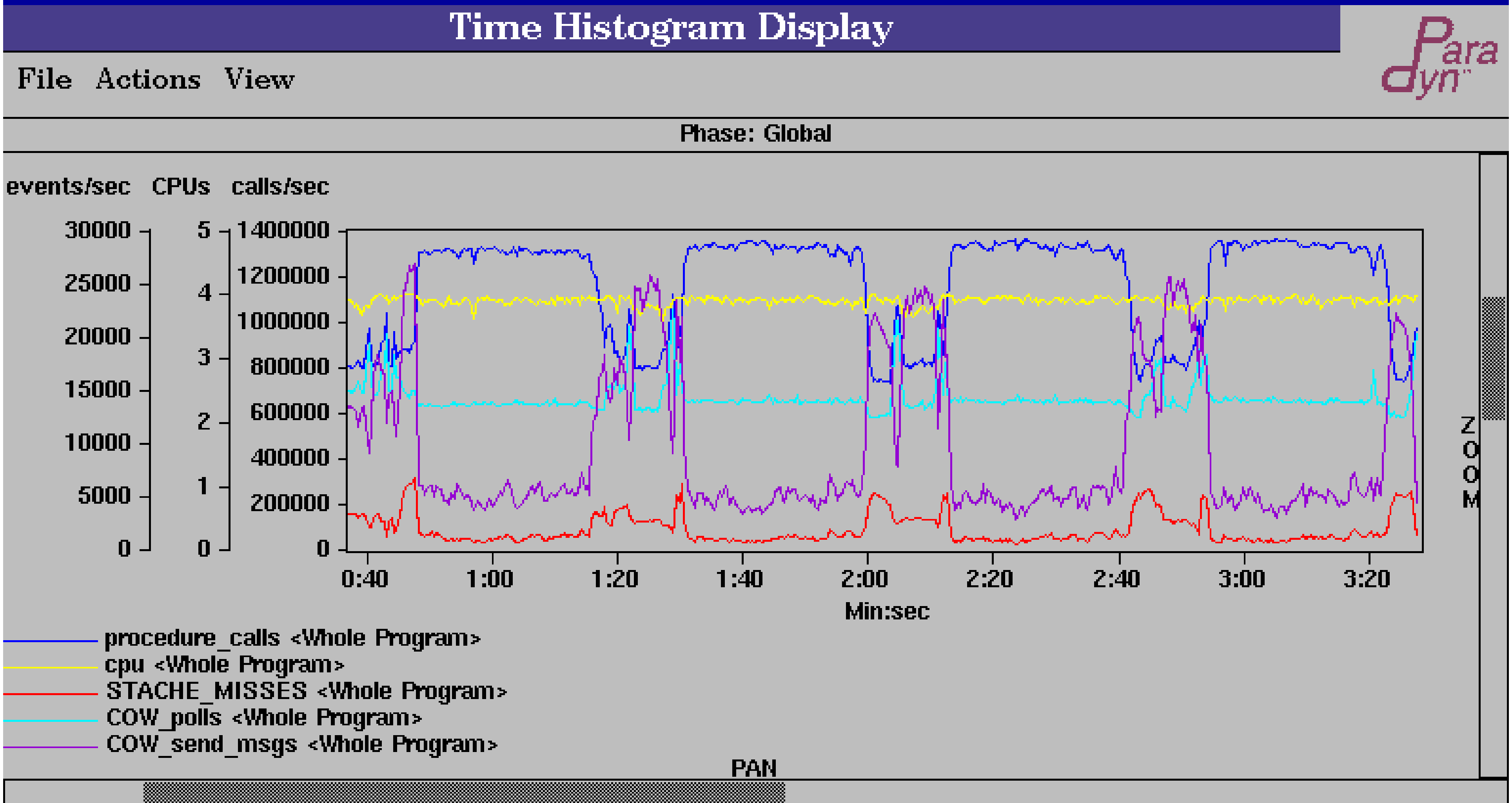
SP2 Hardware Perf Counters (IBM xlc)



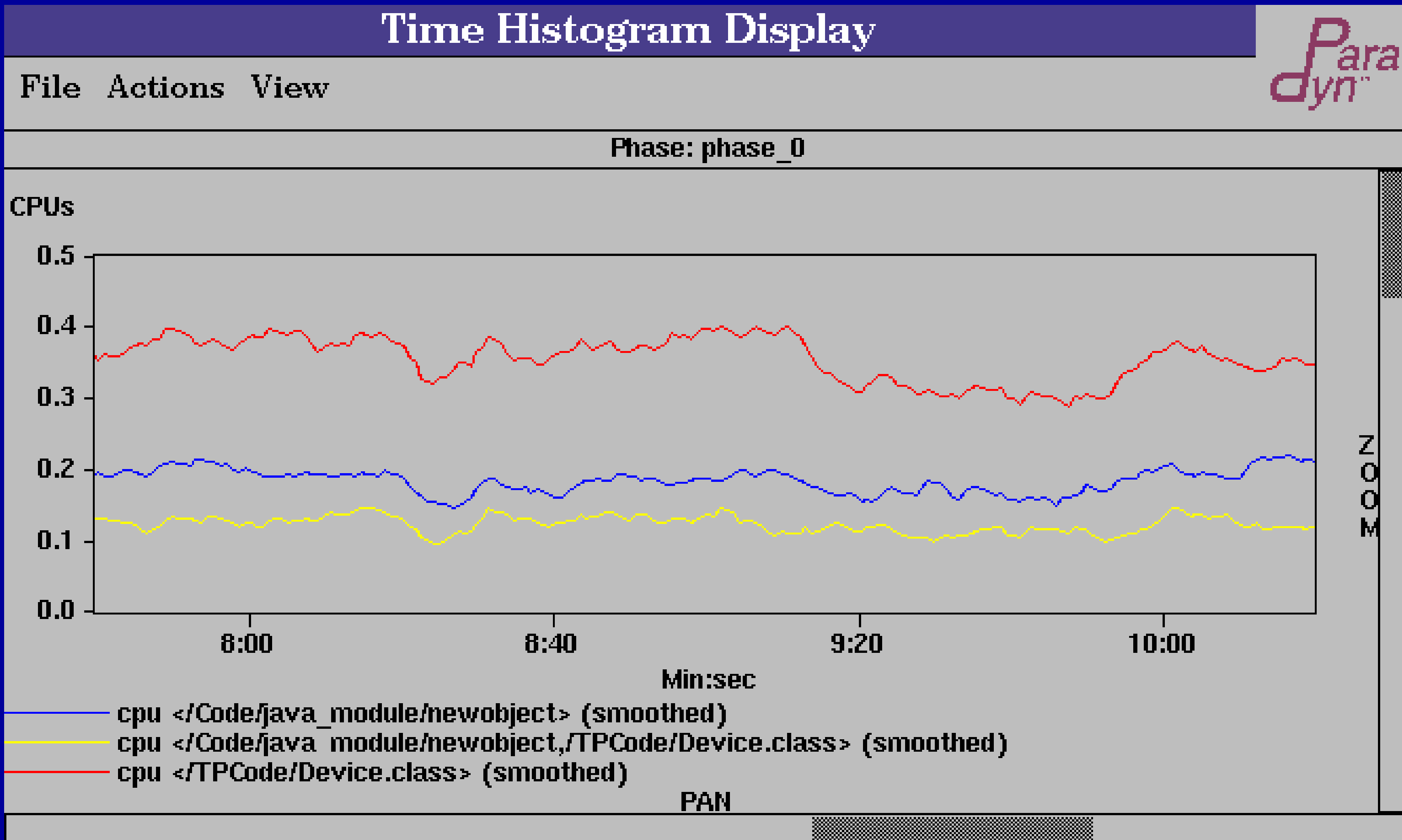
UltraSPARC Hardware Perf Counters



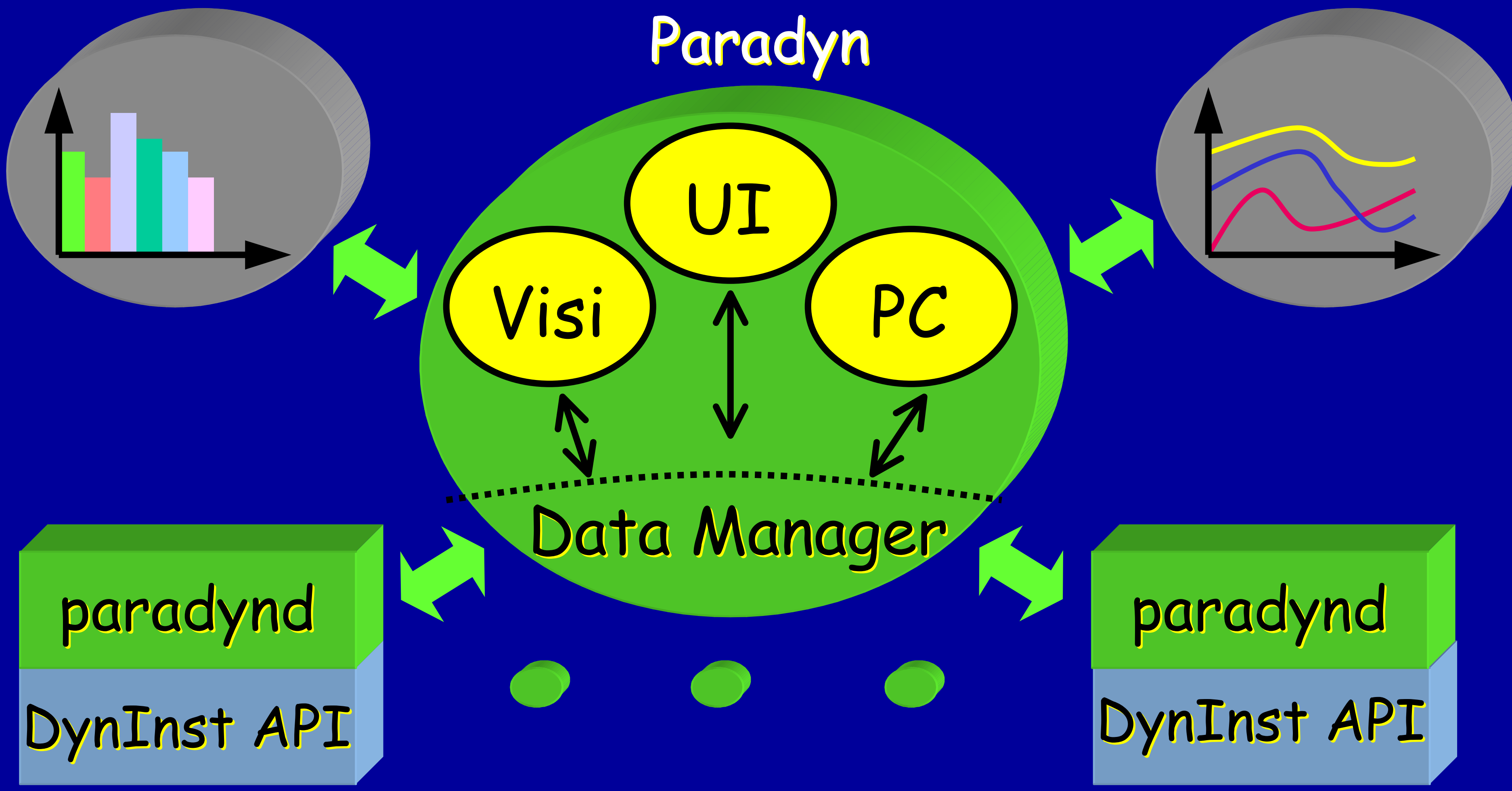
Paradyn Running on Blizzard/Cow (Barnes)



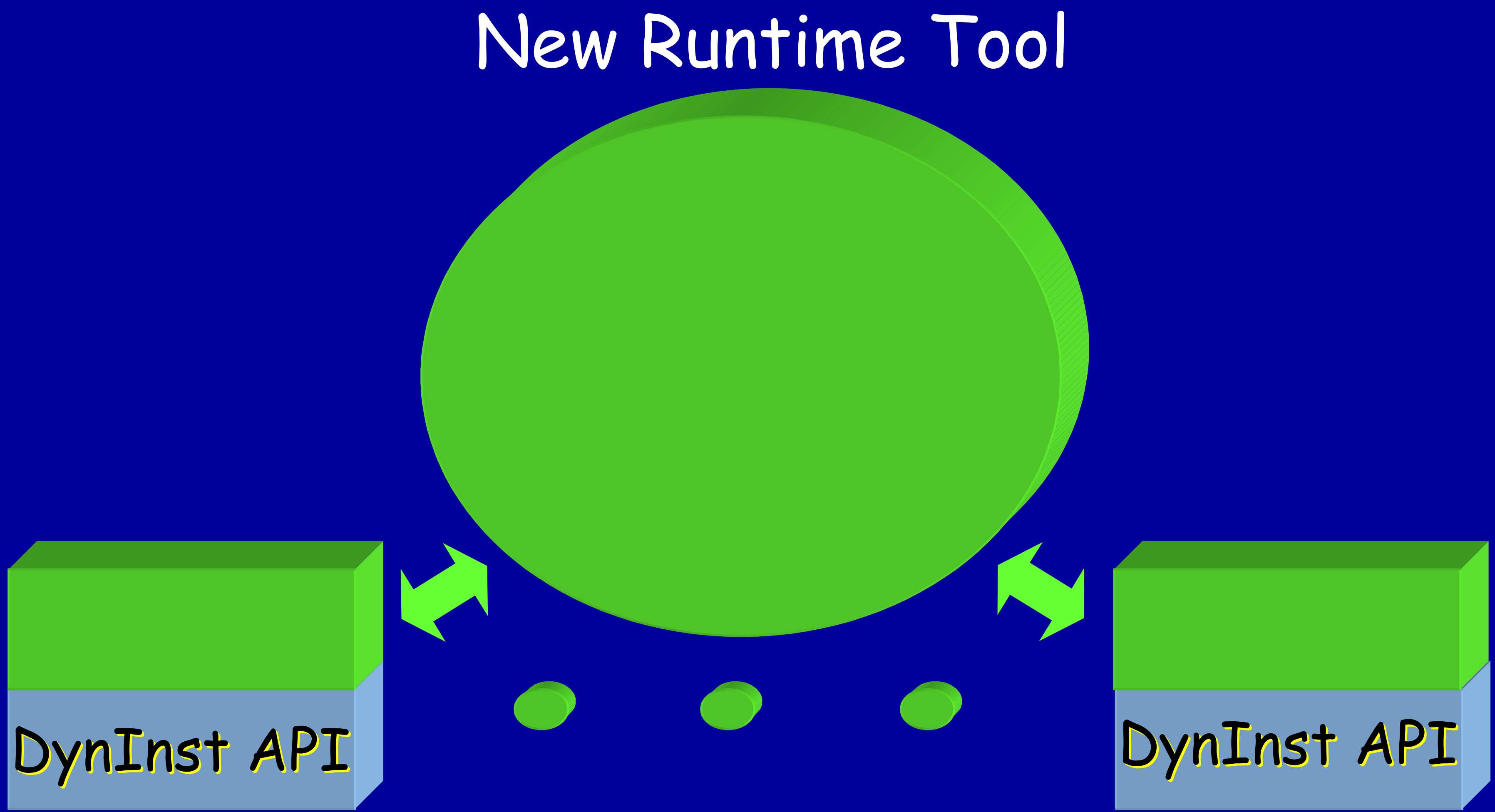
Java Application and VM Profiling



DynInst API: A Common Substrate



DynInst API: A Common Substrate



How to Get a Copy of Paradyn:

Release 3.0 (beta)

- Documentation: Installation Guide, Tutorial, Users Guide, Visi Programmers Guide, Libthread Programmers Guide, Developers Guide, MDL Programmers Guide.
- Free for research use.
- Runs on Solaris (SPARC & x86), NT (x86), Irix, AIX/SP2, Linux (x86), DEC Unix.

http://www.cs.wisc.edu/paradyn

paradyn@cs.wisc.edu

MDL Example: Define Instrumentation Points

```
resource list pvm_sync_ops is procedure
  {
   items {"pvm_send", "pvm_recv"};
   flavor {pvm}
   library true:
}
```

MDL Example: Basic Metric Defn (Sync Wait)

```
metric P_syncWait {
  name "PVM SyncWait";
  units Seconds; unitStyle normalized;
  aggregateOperator agv;
  flavor {pvm}

  constraint functionConstraint;
  constraint moduleConstraint;
  constraint msgTagConstraint;
```

```
metric P_syncWait {
  base is wallTimer {
    foreach func in pvm_sync_ops{
    append preInsn func.entry
     constrained
     (*
       startWallTimer(p_syncWait);
    *)
    prepend postInsn func.entry
     constrained
     (*
    stopWallTimer(p_syncWait)
    *)
  }
}
```

```
constraint functionConstraint
  /Code/Module is counter {
   append preInsn $constraint.entry
      (* funcConstraint = 1; *)
   prepend postInsn $constraint.exit
      (* funcConstraint = 0; *)
   foreach callsite in $constraint.calls {
      append preInsn callsite
         (* funcConstraint = 0; *)
      prepend postInsn callsite
         (* funcConstraint = 1; *)
   }
}
```

```
constraint msgTagConstraint
  /SyncObject/MsgTag is counter {
   foreach func in pvm_sync_ops {
      append preInsn func.entry
       constrained
         (* if ($arg[1] == $constraint)
              msgTagConstraint = 1;
         *)
      prepend postInsn callsite
         (* msgTagConstraint = 0; *)
   }
}
```