PAPI, DynInst and Hardware Performance Analysis Tools

Lawrence Berkeley National Laboratory

Performance Workshop, August 8th, 2003

Philip Mucci, Research Consultant

Innovative Computing Laboratory/UTK

Performance Evaluation Research Center/NERSC/LBNL

pjmucci@lbl.gov

The PAPI Interface

PAPI provides two standardized APIs to access the underlying performance counter hardware

- A low level interface designed for tool developers and expert users.
- The high level interface is for application engineers.

Overview of Talk

- Overview of PAPI
  - Features, Functionality and Usage
  - 2.3.4 Release Status
  - Changes coming in 3.0
- Dynamic Instrumentation
- Performance Analysis Tools
- Trends in the field

Purpose

The purpose of the PAPI project is to design, *standardize* and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.

Motivation

- To increase application performance and system throughput
- To characterize application and system workload on the CPU
- To stimulate performance tool development and research
- To stimulate research on more sophisticated feedback driven compilation techniques

Goals

- Provide a solid foundation for cross platform performance analysis *tools*.
- Loosely *standardize* interface among users, tool developers, vendors and academics.
- Provide implementations for the more popular HPC machines.
- Easy to use, well documented and freely available. (Truly Open Source)

The Fallacy of Good Scalability

- Engineers often report near linear scalability for large parallel application codes to hundreds of processors.
  - Closer inspection reveals poor per processor performance for their problem class.
  - How does one judge performance?
    - Do we have *good* algorithm? (maybe)
    - Do we have a *good* compiler? (?)
    - We need real performance data from the processor!

Peak Performance

- How fast is fast?
  - Single CPU performance in the scientific marketplace often results from good **data locality**, predictable branch behavior, and *instruction pipelines filled with independent instructions*. (Out-of-order scheduling and execution helps the latter 2)
  - Determining the "reuse potential" for a numerically intensive problem often dictates peak performance.
  - Certain ratios of metrics can help diagnose this problem. FP stalls vs. Mem stalls, loads vs. stores, misses vs. references, etc…

HPC Performance Analysis

- Portability of an application is of the utmost concern to most HPC environments.
- No common performance tools except **`prof/gprof`**.
- *Most* commercial tools are based on time.
- HPC have memory and floating point intensive workloads which require detailed analysis.
- Research tools are the most common “solution” among the labs.

Overview of Hardware Counters

- Small number of registers dedicated for performance monitoring functions.
  1. AMD Athlon, 4 counters
  2. Pentium <= III, 2 counters
  3. Pentium IV, 18 counters
  4. IA64, 4 counters
  5. Alpha 21x64, 2 counters
  - Power 3, 8 counters
  - Power 4, 8 counters
  - UltraSparc II, 2 counters
  - MIPS R14K, 2 counters

Power 4 Module and Block Diagram

Figure 1: POWER4 Chip Logical View

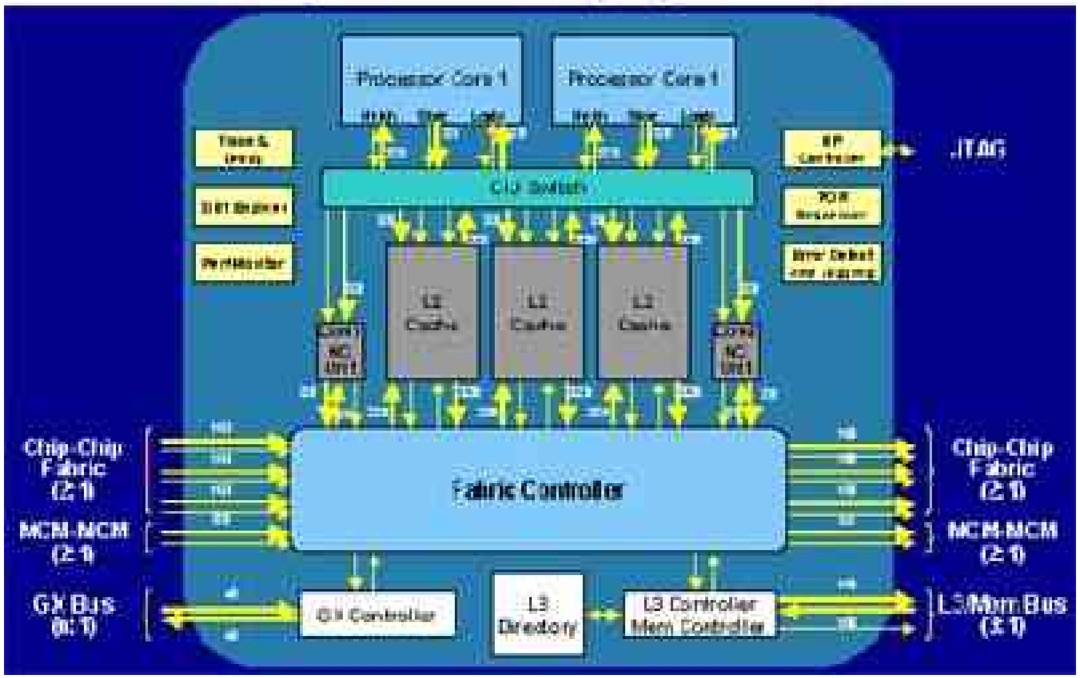
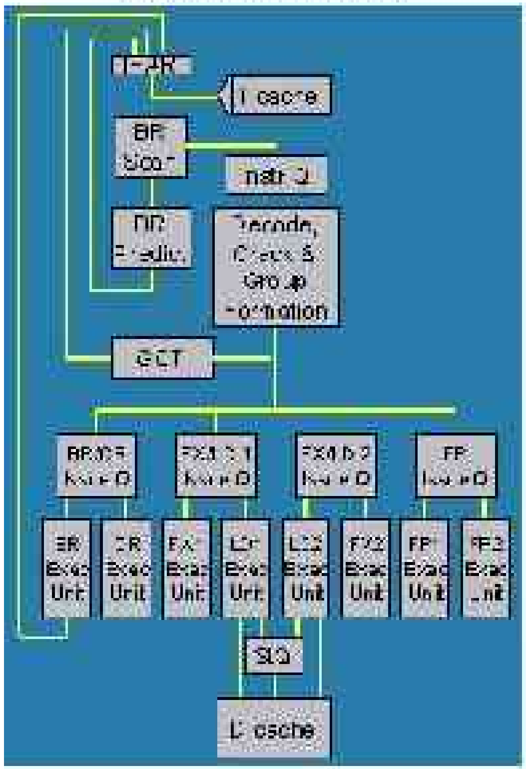
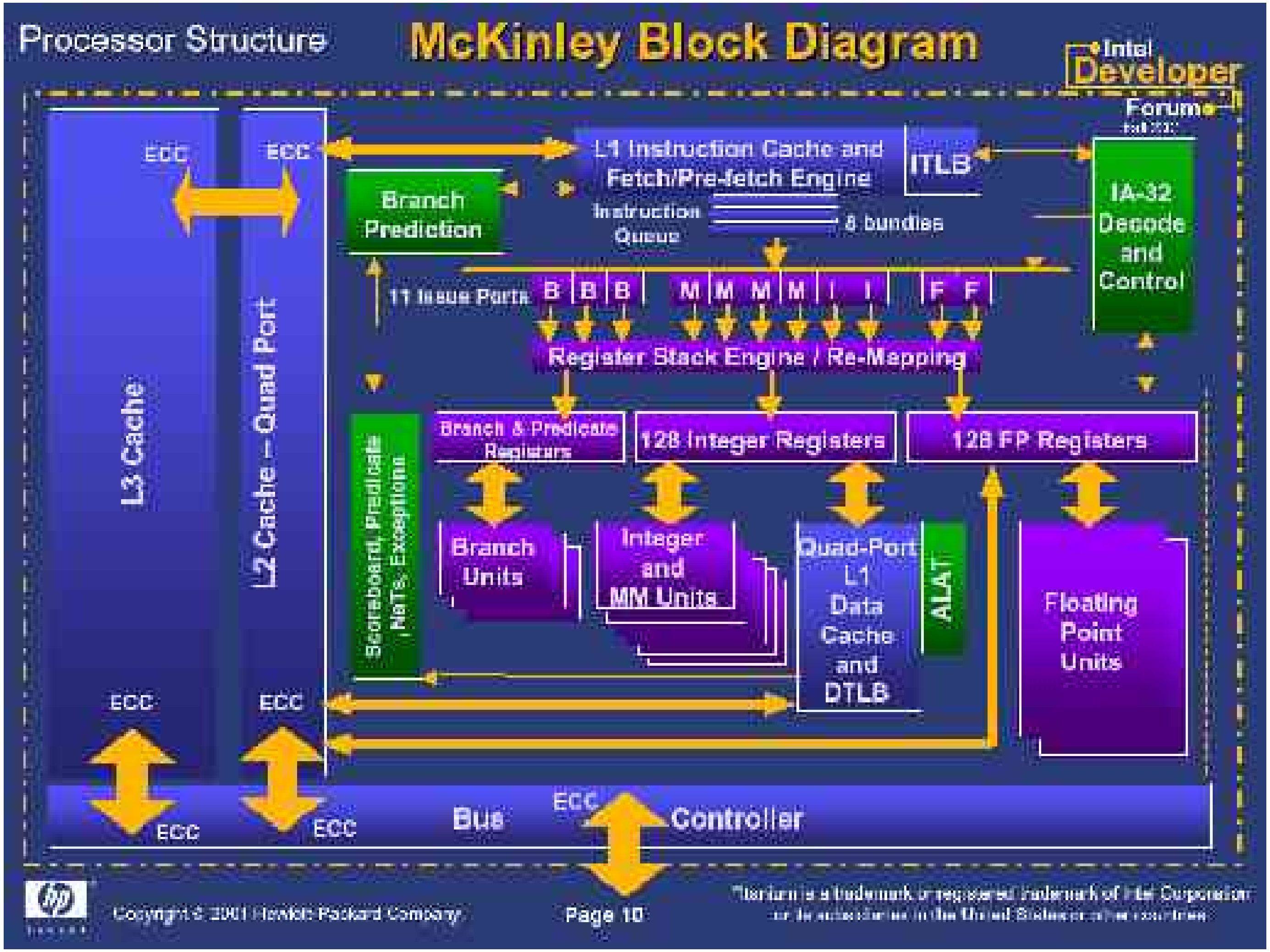


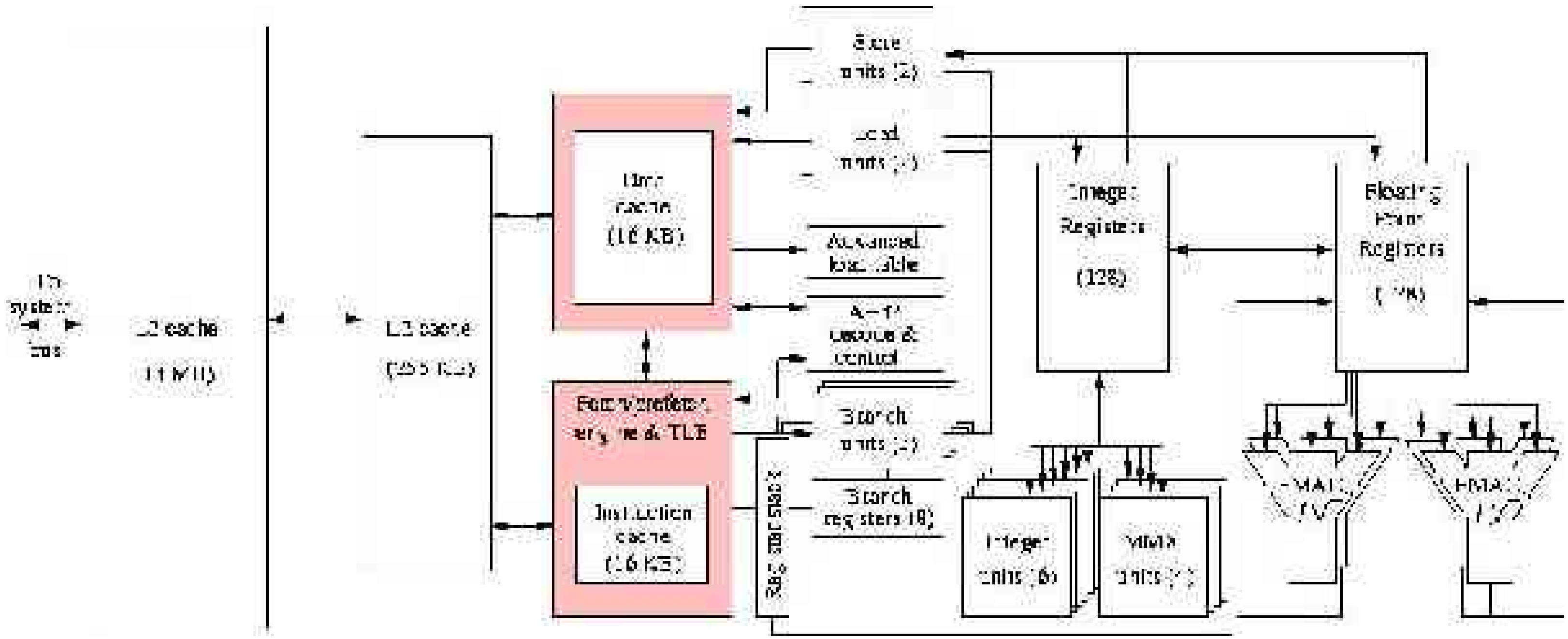
Figure 2: POWER4 Core



Itanium 2 Block Diagram

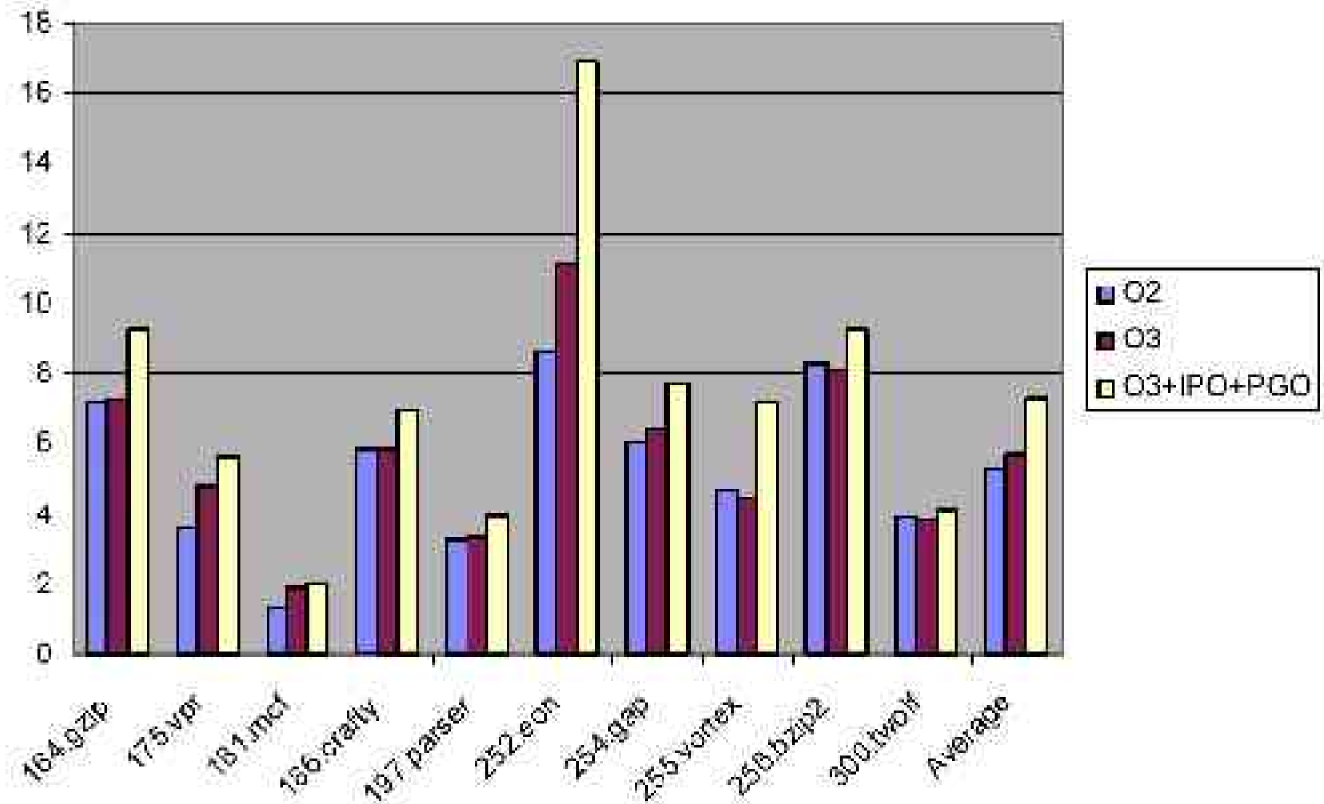


Itanium 2 Block Diagram

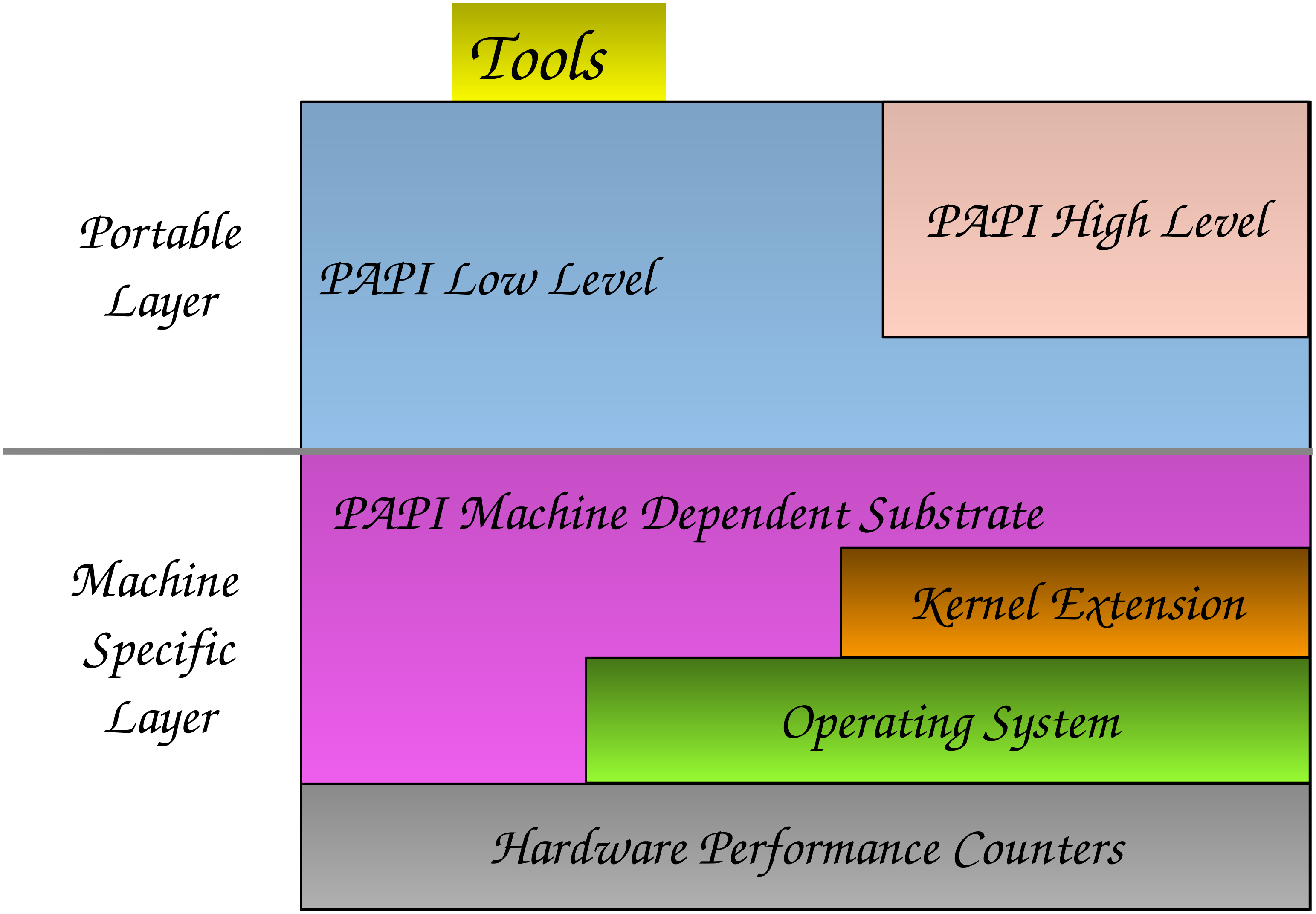


Importance of Optimization

Example: Speed up from Static Compiler Optimization on Itanium-I in 2002 (SpecInt)



PAPI Implementation



Preset Events

- Proposed standard set of event names deemed most relevant for application performance tuning
- No standardization of the actual definition
- Mapped to native events on a given platform

Preset Events 2

- PAPI supports approximately 100 preset events.
  - Preset events are mappings from symbolic names to machine specific definitions for a particular hardware event.
    - Example: **PAPI_TOT_CYC**
  - PAPI also supports presets that may be derived from the underlying hardware metrics
    - Example: **PAPI_L1_DCM**

Sample Preset Listing

```
Test case 8: Available events and hardware information.
-------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (-1)
Model string and code    : Celeron (Mendocino) (6)
CPU revision             : 10.000000
CPU Megahertz            : 366.504944
-------------------------------------------------------------------------
Name          Code          Derived  Description (Note)
PAPI_L1_DCM  0x80000000   No       Level 1 data cache misses
PAPI_L1_ICM  0x80000001   No       Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002   No       Level 2 data cache misses
PAPI_L2_ICM  0x80000003   No       Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004   No       Level 3 data cache misses
PAPI_L3_ICM  0x80000005   No       Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006   Yes      Level 1 cache misses
PAPI_L2_TCM  0x80000007   No       Level 2 cache misses
PAPI_L3_TCM  0x80000008   No       Level 3 cache misses
PAPI_CA_SNP  0x80000009   No       Requests for a snoop
PAPI_CA_SHR  0x8000000a   No       Requests for shared cache line
PAPI_CA_CLN  0x8000000b   No       Requests for clean cache line
PAPI_CA_INV  0x8000000c   No       Requests for cache line inv.
```

Native Events

- PAPI supports native events:
  - An event countable by the CPU can be counted even if there is no matching preset PAPI event.
  - The developer uses the same API as when setting up a preset event, but a CPU-specific bit pattern is used instead of the PAPI event definition.

High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- As easy to use as the calls present in IRIX.
- Requires no setup code
- Restrictions:
  - Only PAPI preset events may be used
  - Not thread safe (in PAPI 2.3.4)

High-level API Calls

- **PAPI_num_counters()**
- **PAPI_start_counters(int *cntrs, int alen)**
- **PAPI_stop_counters(long_long *vals, int alen)**
- **PAPI_accum_counters(long_long *vals, int alen)**
- **PAPI_read_counters(long_long *vals, int alen)**
- **PAPI_flops(float *rtime, float *ptime, long_long *flpins, float *mflops)**

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Approximately 60 functions

  (http://icl.cs.utk.edu/projects/papi/files/html_man/papi.html#4)
- Thread-safe for all 1:1 thread libraries. (Native, OpenMP, Pthreads, etc...)
- Supports both presets and native events

Low-level Functionality

- API Calls for:
  - Counter multiplexing
  - Callbacks on user defined overflow value
  - SVR4 compatible profiling
  - Processor information
  - Address space information
  - Static and dynamic memory information
  - Accurate and low latency timing functions
  - Hardware event inquiry functions
  - Eventset management functions
  - Simple locking operations

PAPI and Multiplexing

- Multiplexing allows simultaneous use of more counters than are supported by the hardware.
  - This is accomplished through timesharing the counter hardware and extrapolating the results.
- Users can enable multiplexing with one API call and then use PAPI normally.

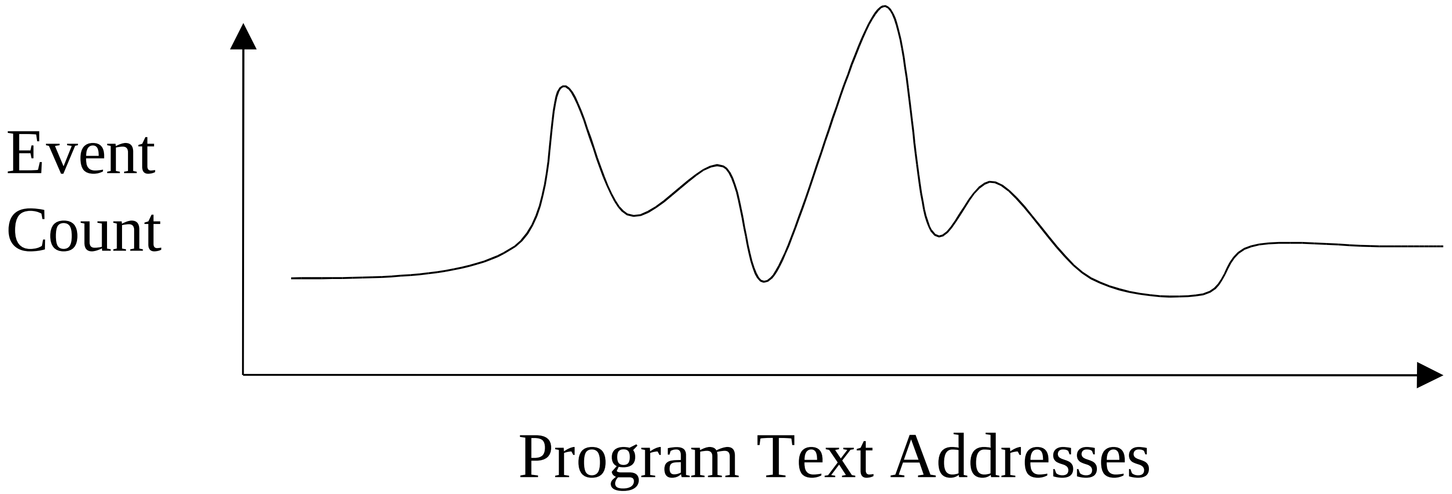
Interrupts on Counter Overflow

- PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.
- For systems that do not support counter overflow at the hardware level, PAPI emulates this in software at the user level.
  - Code must run a reasonable length of time.

Hardware Profiling

- On overflow of hardware counter, dispatch a signal/interrupt.
- Get the address at which the code was interrupted.
- Store counts of interrupts for *each* address.
- Vendor/GNU **prof** and **gprof** (**-pg** and **–p** compiler options) use interval timers.

Results of Statistical Profiling



- The result: A probabilistic distribution of where the code spent its time and why.

For More Information

- http://icl.cs.utk.edu/projects/papi/
  - Software and documentation
  - Reference materials
  - Papers and presentations
  - Third-party tools
  - Mailing lists

PAPI 2.3.4 Release

- Supported Platforms
  - IBM 604, 604e, Power 3, 4
  - Intel x86, Pentium IV
  - Intel Itanium I, II
  - Sun UltraSparc I/II/III
  - SGI R10K/R12K/R14K
  - Compaq Alpha 21164/21264 with DADD/DCPI
  - Cray T3E
  - AMD Opteron
  - Windows/x86 (not PIV)
- Enhancements
  - Static/dynamic memory info
  - Multiplexing improvements
  - Lots of bug fixes

PAPI 3.0

- Using lessons learned from years earlier
  - Substrate code: 90% used only 10% of the time
- Complete internal redesign for:
  - Efficiency
  - Robustness
  - Feature Set
  - Elegance
  - Portability

Some PAPI 3.0 Features

- Multiway multiplexing
  - Use all available counter registers instead of one per time slice.
- Superb performance
  - Example: On Pentium 4, a PAPI_read() costs 230 cycles. (Register read costs 100 cycles)
- Full native and programmable event support
  - Thresholding
  - Instruction matching
  - Per event counting domains

PAPI 3.0 Features 2

- Third-party interface
  - Allows PAPI to control counters in other processes
- Internal timer/signal/thread abstractions
  - Support signal forwarding
- Additional internal layered API to support robust extensions

PAPI 3.0 Features 3

- Advanced profiling interface
  – Support profiling on multiple counters
  – Support hardware or operating system assisted profiling
- New sampling interface
  – P4, IA64 provide Event Address Registers of BTB misses, Cache misses, TLB misses, etc…
- Revised memory usage API
  – Process footprint

PAPI 3.0 Features 4

- System-wide and process wide counting implementation
- Expanded high level API
  - Thread safe
- New language bindings:
  - Java
  - Lisp
  - Matlab

PAPI 3.0 Release

- Initial release expected around SC 2003 (limited scope)
- Additional platforms will be added as they come available:
  - Cray X-1 (partially complete)
  - Nec SX-6
  - Blue Gene (BG/L)

DyninstAPI

- API for runtime code patching
  - new code can be added to a program while it executes
  - permits instrumentation and modification of programs
- Provides processor independent abstractions
  - same patching can be applied to multiple systems
- Includes meta-instrumentation
  - tracks overhead on inserted code

IBM's DPCL vs. DynInst

- Parallel framework based on early DynInst
- Async/Sync operation
- Functions for getting data back to tool
- Integrated with POE

- Available on all HPC platforms (and Windows)
- Breakpoints
- Arbitrary ins. points
- Full Loop, CFG and Basic Block decoding

A Brief History of Dynamic Instrumentation

- Popularized by James Larus with EEL: An Executable Editor Library at U. Wisc.
  - http://www.cs.wisc.edu/~larus/eel.html
- Technology matured by Dr. Bart Miller and (now Dr.) Jeff Hollingsworth at U. Wisc.
  - DynInst Project at U. Maryland
    - http://www.dyninst.org/
  - IBM's DPCL: A Distributed DynInst
    - http://oss.software.ibm.com/dpcl/

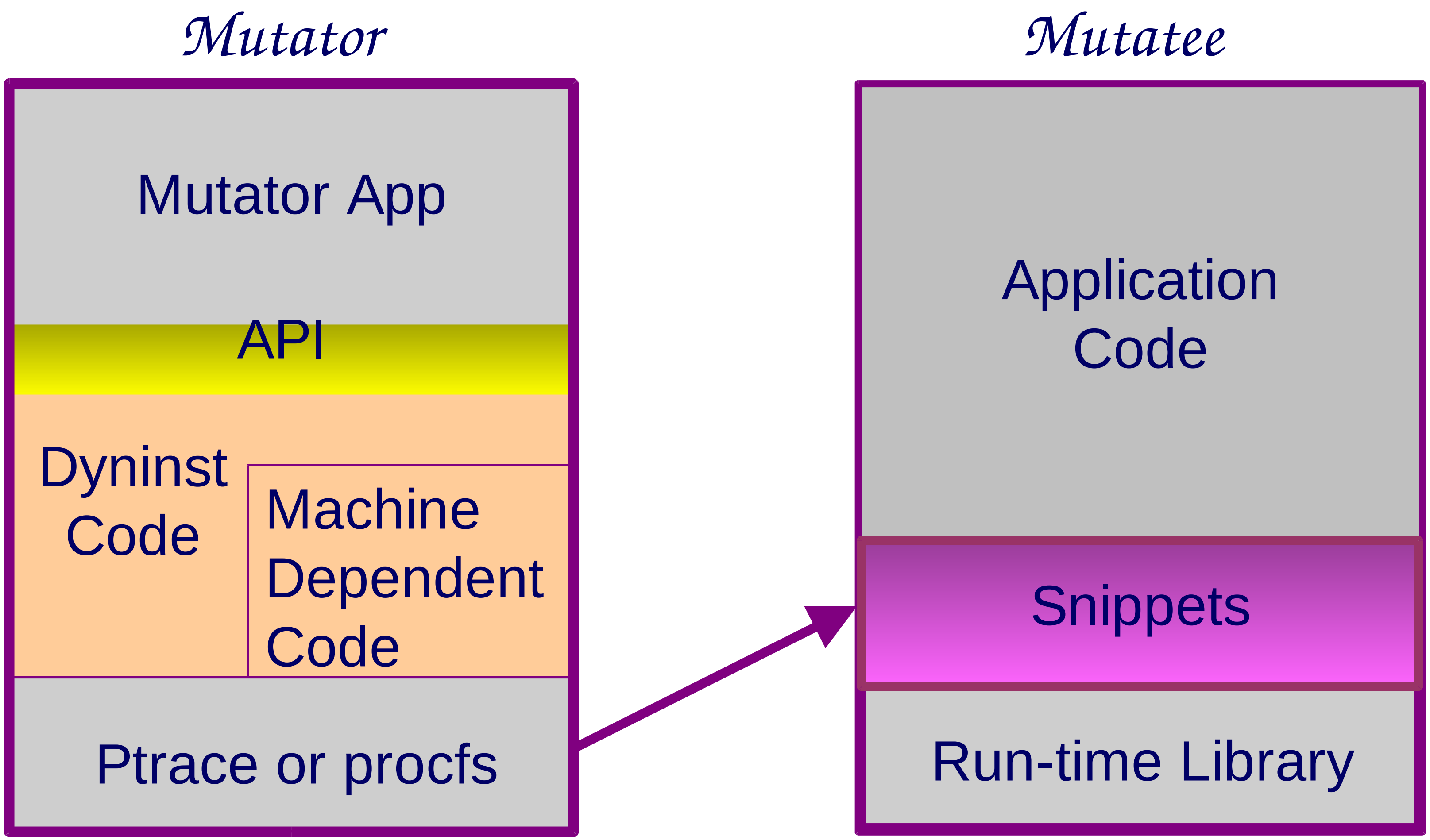
Why Runtime Code Patching?

- Performance measurement
  - Recording application behavior
- Correctness debugging
  - Fast conditional breakpoints
  - Data breakpoints
- Execution driven simulation
  - Architecture studies
- Testing
  - Code coverage testing
  - Forcing hard to execute paths to be taken

Advantages of Runtime Code Patching

- No forethought needed
  - No user inserted probes
  - No special compiling or linking
  - Start anytime during execution
- Only insert code when needed
  - No wasted checks for “disabled” code
  - Can add new code during execution

Structure of the Dyninst Library



API Library

- Provides:
  - Functions for control of mutatee
  - Runtime code generation
  - Information about mutatee

- A set of C++ classes
  - Bpatch_Thread
  - BPatch_image
  - BPatch_snippet
  - BPatch_variableExpr

Representing Code Snippets

- Platform Independent Representation
  - Same code can be inserted into apps on any system
- Simple Abstract Syntax Tree
  - Can refer to application state (variables & params)
  - Includes simple looping construct
  - Permits calls to application subroutines
- Type Checking
  - Ensures that snippets are type compatible
  - Based on structural equivalence
    - allows flexibility when adding new code

Type Support in Dyninst

- Access to local (stack) variables
- Complex types
  - non-integer scalars
  - structures
  - arrays
  - Fortran common blocks
- Correctness debugging
  - print contents of data structures

Fine-Grained Instrumentaton

- New classes added to dyninstAPI
  - BPatch_basicBlock
  - BPatch_sourceBlock
  - BPatch_flowGraph
- Arbitrary Instrumentation points
  - Conservative base trampoline
- Base trampoline deletion

Arbitrary Instrumentation Points

- Code Coverage needs basic block level instrumentation
  - dyninstAPI used to support function level instrumentation for sparc-solaris
  - added arbitrary instrumentation points for SPARC
- More state must be maintained in base trampolines
  - save/restore condition codes before/after arbitrary instrumentation points
  - Sparc arch supports user mode condition-code write/read for version v8plus and later

Memory Instrumentation

- Dynamic memory access instrumentation
  – collect low level memory accesses
  – with the flexibility of dynamic instrumentation
- Possible applications
  – offline performance analysis (Sigma etc.)
  – online optimization
  – tools to catch memory errors

Memory Instrumentation Features

- Finding memory access instructions
  - loads, stores, prefetches
- Builds on Arbitrary Instrumentation
- Decoded instruction information
  - type of instruction
  - constants and registers involved in computing
    - the effective address
    - the number of bytes moved
  - available in the mutator before execution
- Memory access snippets
  - effective address in process space
  - byte count
  - available in mutatee at execution time

Saving Binary Modifications

- Re-running with same modified code requires
  - Parse debug symbols
  - Two processes
  - DyninstAPI shared library
  - Time to re-insert instrumentation
- ***Not*** a checkpointing mechanism
  - Mutated binary begins at the top of main()
  - Data initialized as in original binary

Dyninst Status

- Supported platforms
  - SPARC (Solaris)
  - x86 (Solaris, Linux, NT)
  - Alpha (Tru64 UNIX)
  - MIPS (IRIX)
  - Power/PowerPC (AIX)

- Software available on the web
  - **`http://www.dyninst.org`**
  - Includes TCL command tool
  - Over 250 sites have downloaded

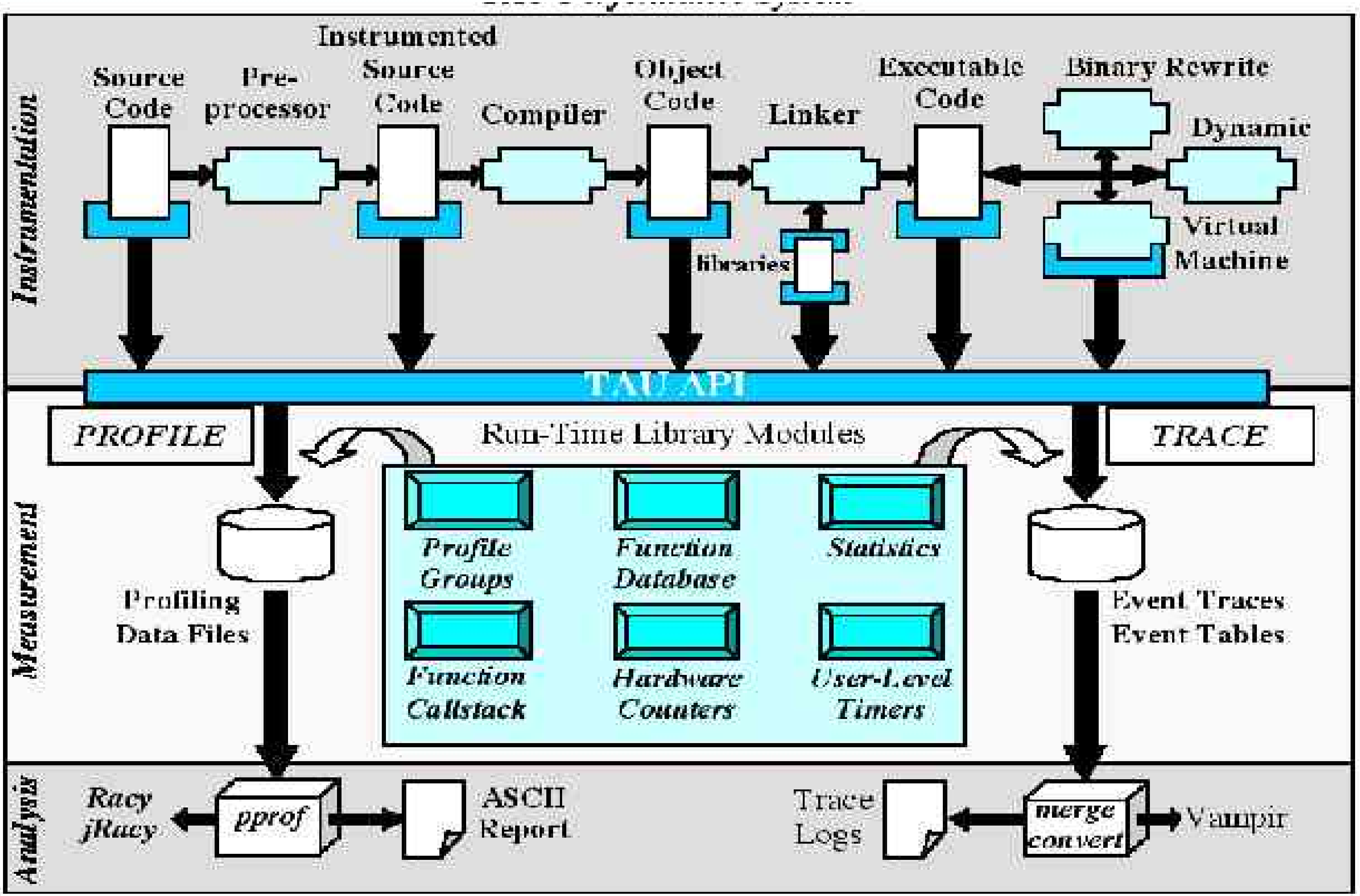
Paradyn from U. Wisconsin

- From Bart Miller Group
- Dynamic discovery of bottlenecks based on testing hypotheses with dynamic instrumentation
- Supports all forms of parallelism
- Different real-time visualization plugins
- Working with PAPI team to discover bottlenecks based on hardware metrics
- Just released version 4.0
- **http://www.paradyn.org**

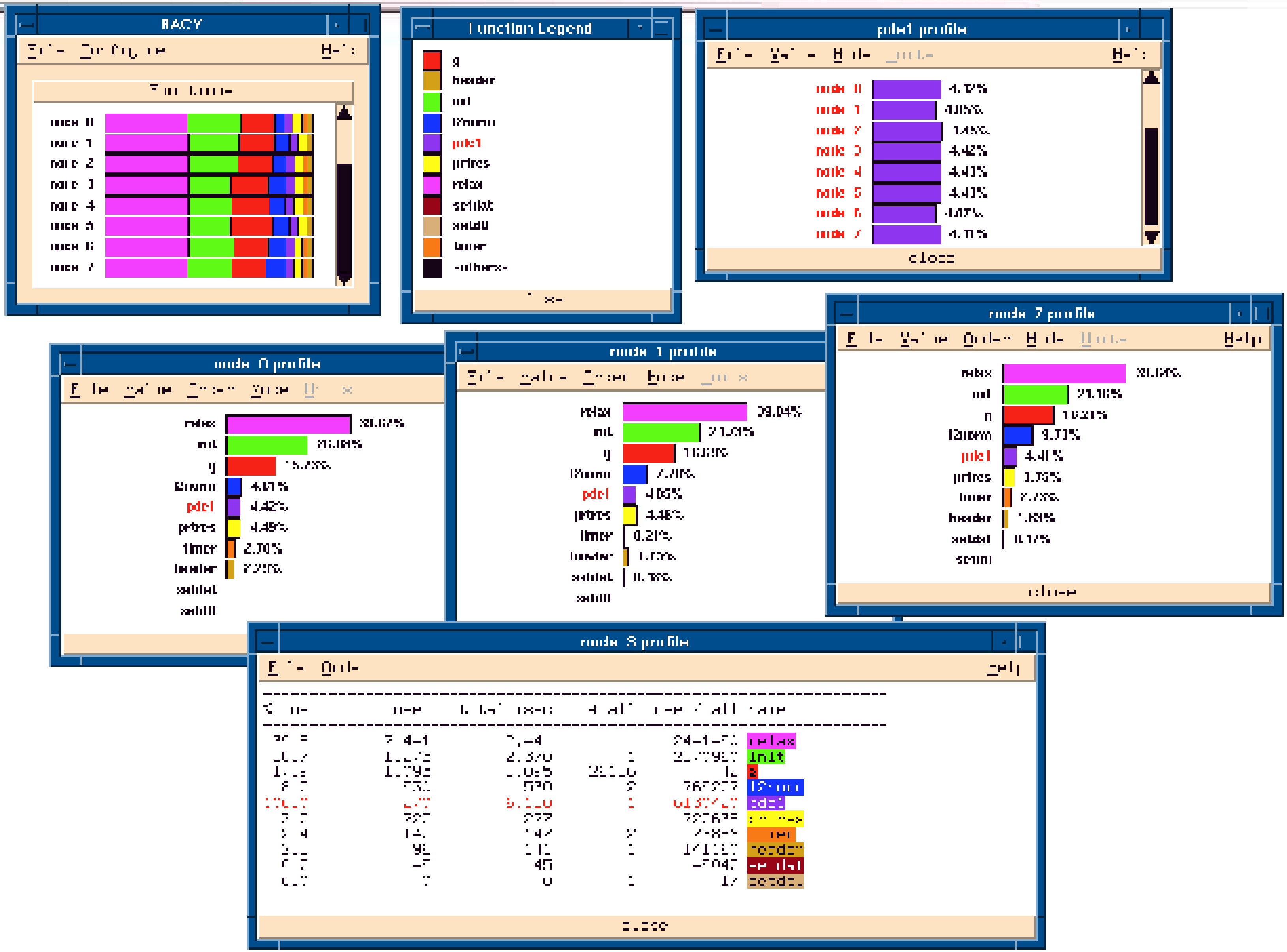
TAU from U. Oregon

- From Allen Malony's Group
- Source or binary based instrumentation
- Supports all forms of parallelism
- Integration with Vampir for trace visualization of MPI, OpenMP and both
- **http://www.cs.uoregon.edu/research/paracomp/tau**

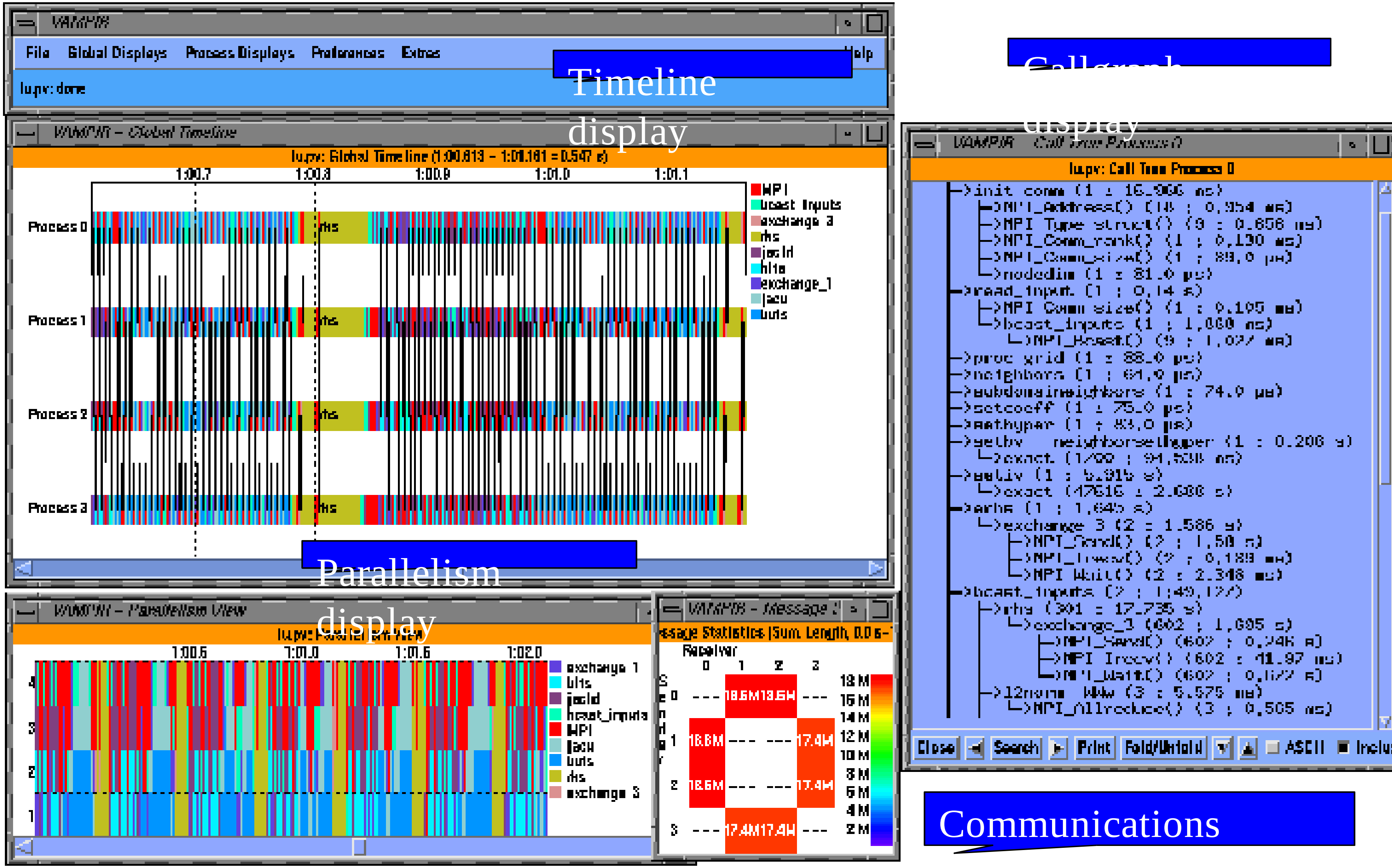
TAU System Architecture



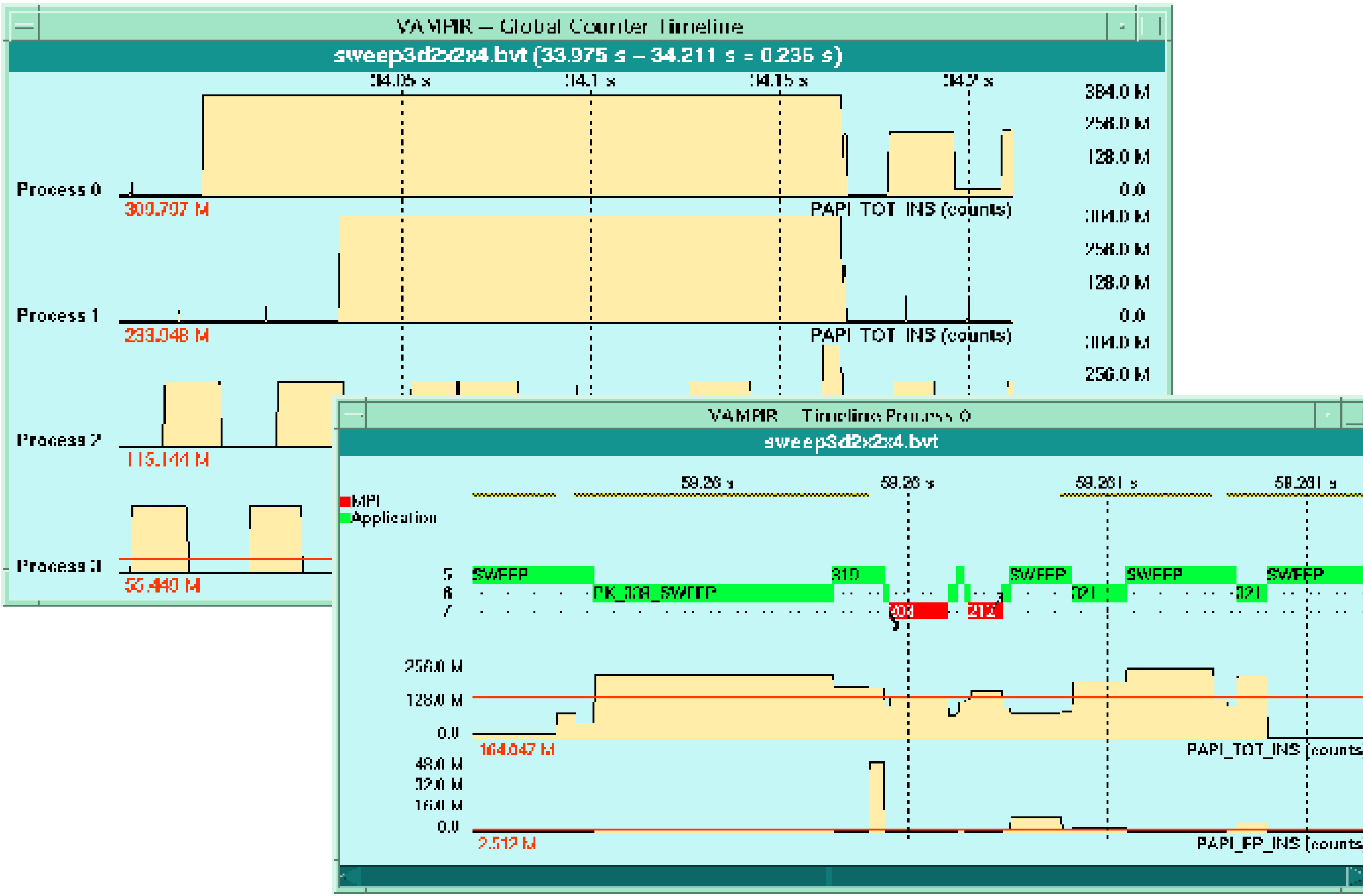
TAU/ParaProf Screenshot



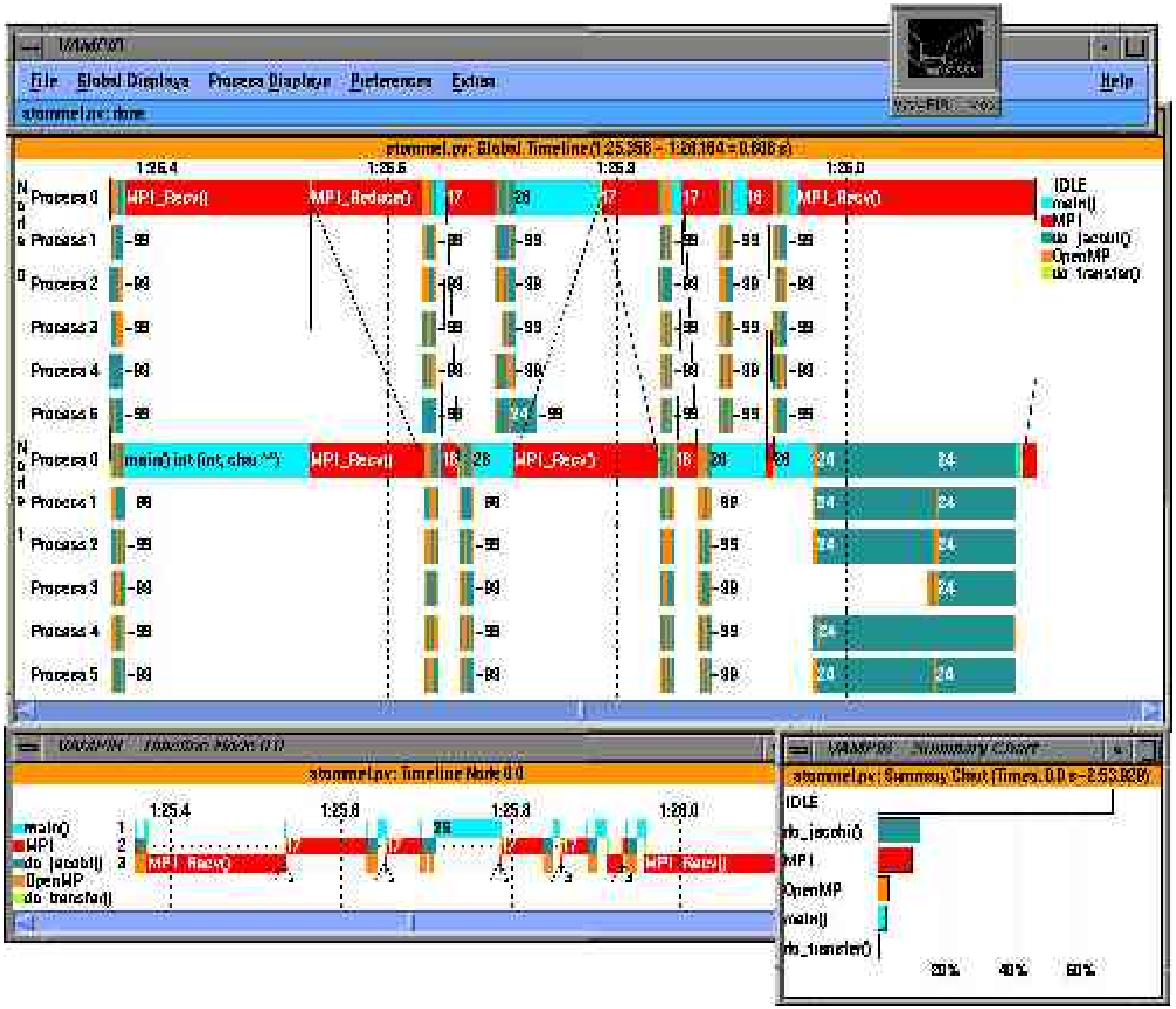
Vampir (NAS Parallel Benchmark – LU)



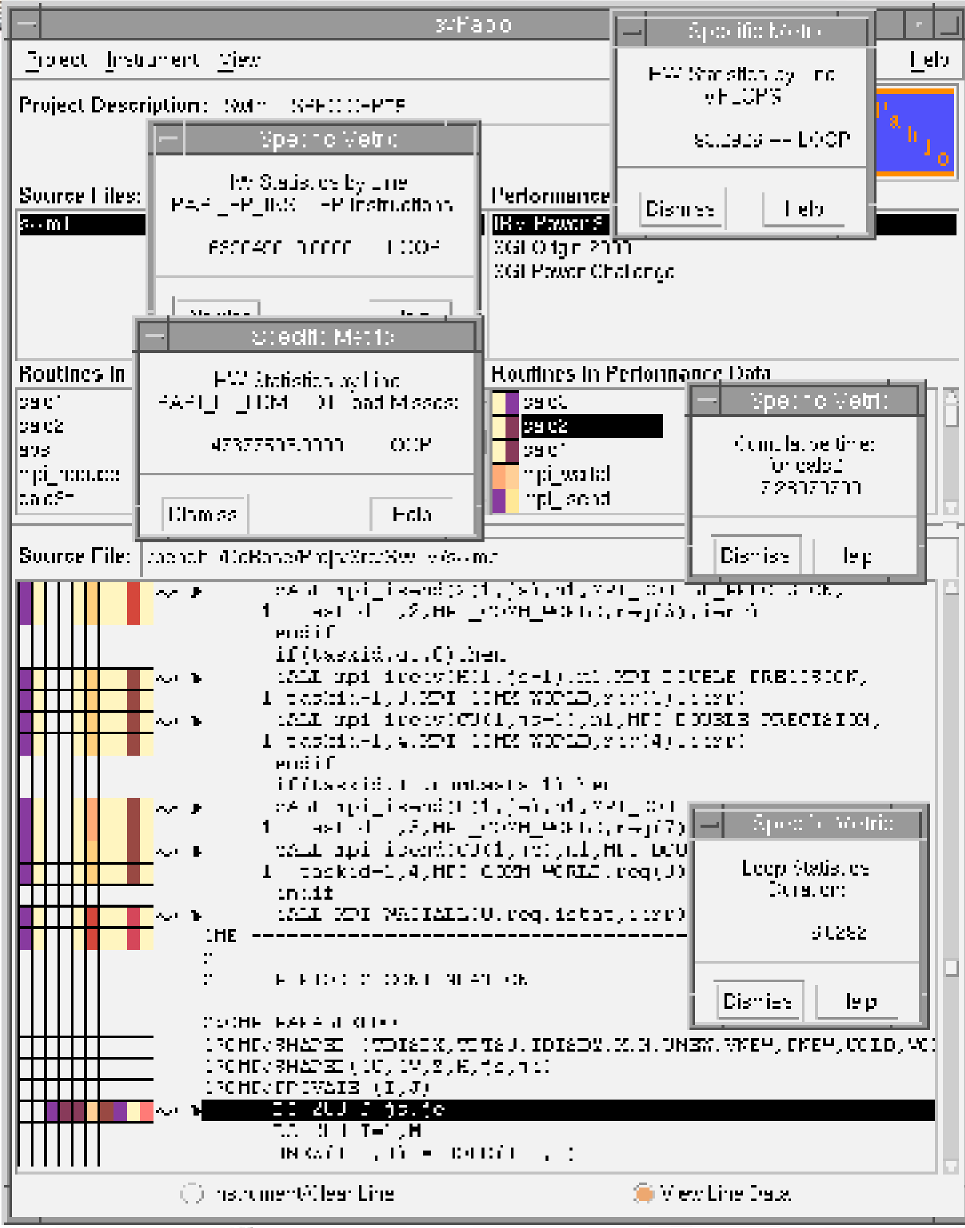
Vampir: PAPI Counter Traces



TAU OpenMP+MPI Vampir Visualization

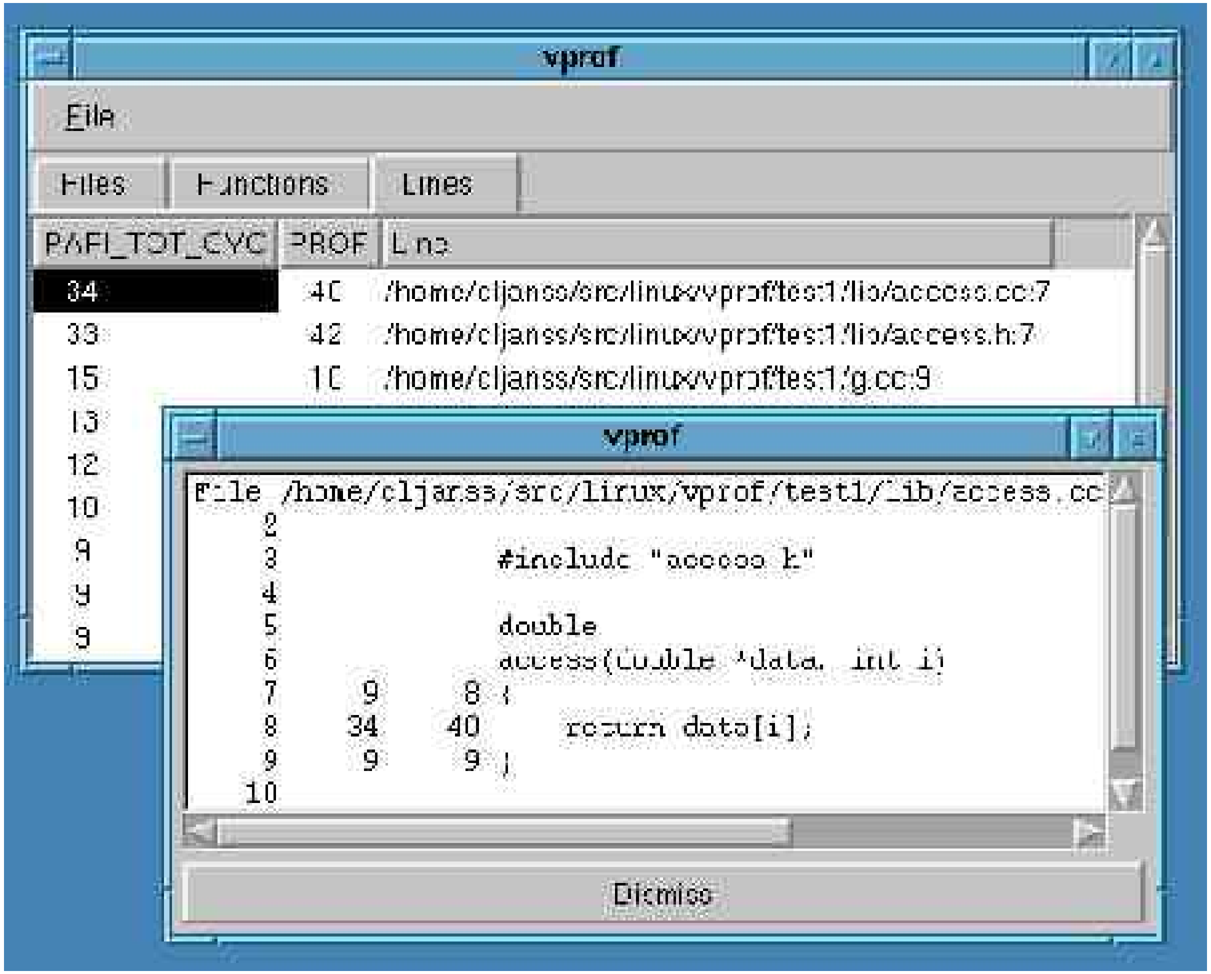


SvPablo from UIUC



- Source based instrumentation of loops and function calls
- Supports serial and MPI jobs
- Freely available
- Rough F90 parser

Vprof from Sandia National Laboratory



- Based on statistical sampling of the hardware counters
- Must instrument the source
- Ported to other architectures for generalized use
- Parallel codes with some modification
- Not actively supported

http://aros.ca.sandia.gov/~cljanss/perf/vprof

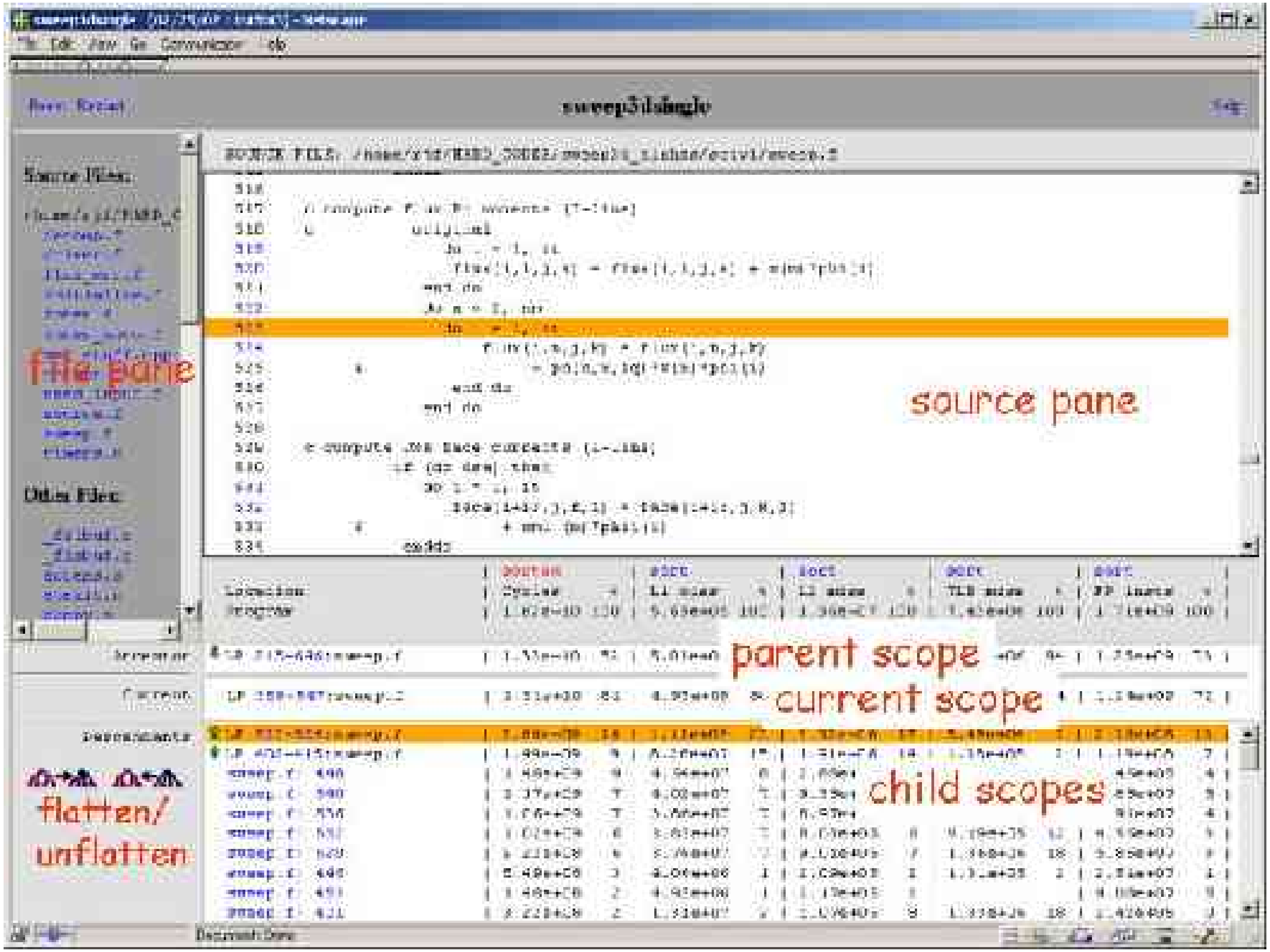
HPCToolkit from Rice University

- Tools for:
  - Collecting raw statistical profiles
  - Conversion of profiles into platform independent XML
  - Synthesizing browsable representations that correlate metrics with source code
- http://www.hipersoft.rice.edu/hpctoolkit

HPCToolkit Tools

- Collection: papirun/hvprof, equivalent to SGI's "ssrun"
- Loop/CFG recovery from binary: bloop
- Data formatting: papiprof
- Data display and exploration: hpcview
- Call stack profiles: csprof
- Data is aggregated into an XML database
- HPCView is a Java applet that generates dynamic HTML

HPCView Screenshot



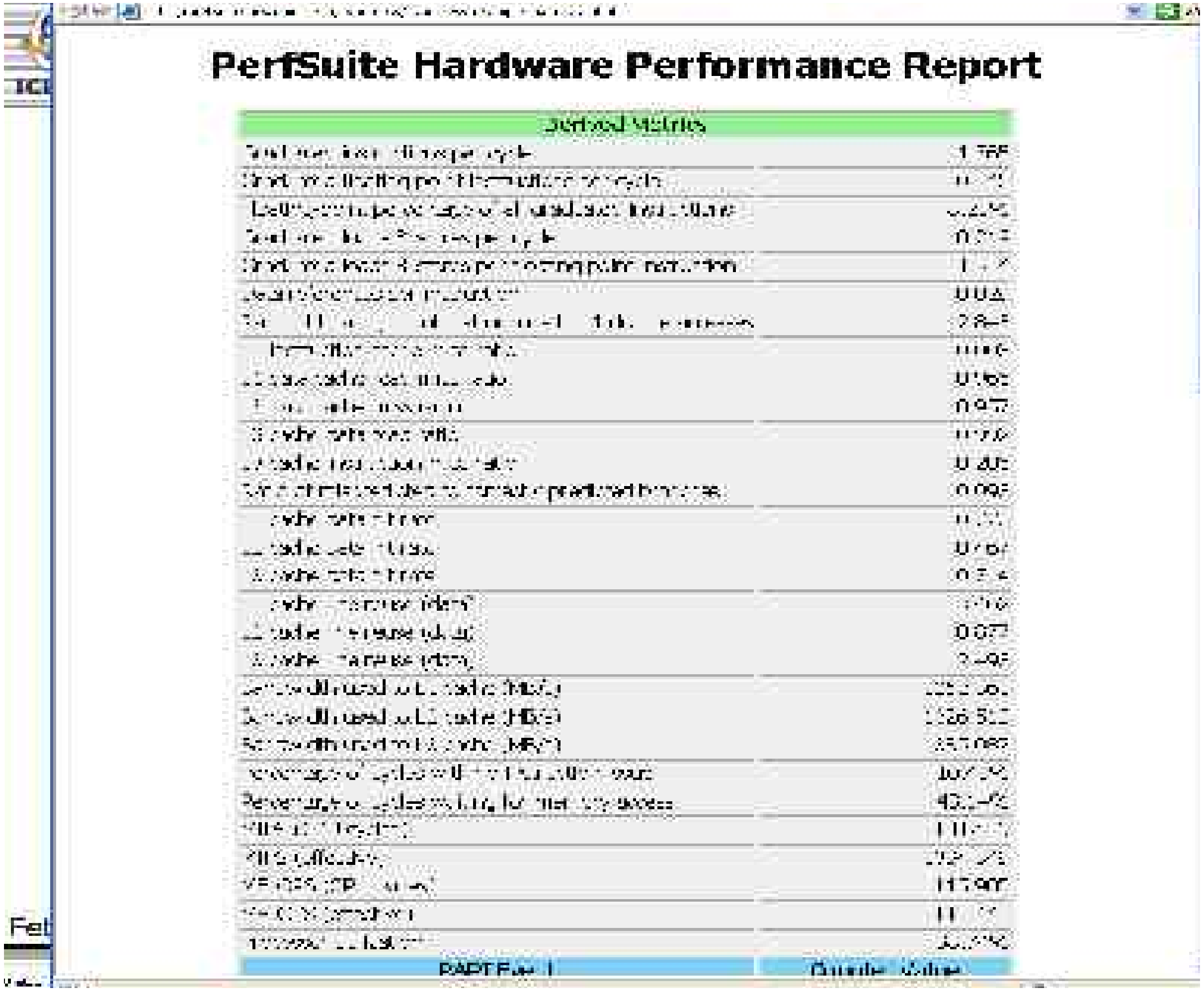
PerfSuite from NCSA

- Libraries and tools for machine information, memory information, aggregate counts, derived metrics and statistical profiles
- Targeted for x86 and IA64 systems
- http://perfsuite.ncsa.uiuc.edu

PerfSuite Tools

- psinv: Gather information on a processor and the PAPI events it supports
- psrun: Collection of aggregate/derived counts or statistical profiles of unmodified binaries
- psprocess: Formatting and output of psrun data into text or HTML

Psprocess Example Output



Psrun Statistical Profile Example Output

Function Summary

Samples	Self %	Total %	Function
1839543	35.01%	35.01%	inl3130
541829	10.31%	45.32%	ns5_core
389741	7.42%	52.74%	inl0100
355349	6.76%	59.51%	spread_q_bsplines
213172	4.06%	63.56%	gather_f_bsplines
200546	3.82%	67.38%	do_longrange
182691	3.48%	70.86%	make_bsplines
149924	2.85%	73.71%	ewald_LRcorrection
112883	2.15%	75.86%	inl3100
105317	2.00%	77.86%	solve_pme
92257	1.76%	79.62%	flincs

PerfSuite Libraries

- libperfsuite: Provides simple wrappers for machine information, process memory usage and high-precision timing
- libpshwpc: Provides simple wrappers that are used to collect hardware performance data

Libpshwpc Example

```
  program mxm
 include 'fperfsuite.h'

c Initialize libpshwpc

      call PSF_hwpc_init(ierr)

c Start performance counting using libpshwpc

      call PSF_hwpc_start(ierr)

c Stop hardware performance counting and write the
c results to a file named 'perf.XXXXX' (XXXXX will be
c replaced by the process ID of the program)

      call PSF_hwpc_stop('perf', ierr)

c Shutdown use of libpshwpc and the underlying libraries

      call PSF_hwpc_shutdown(ierr)
```

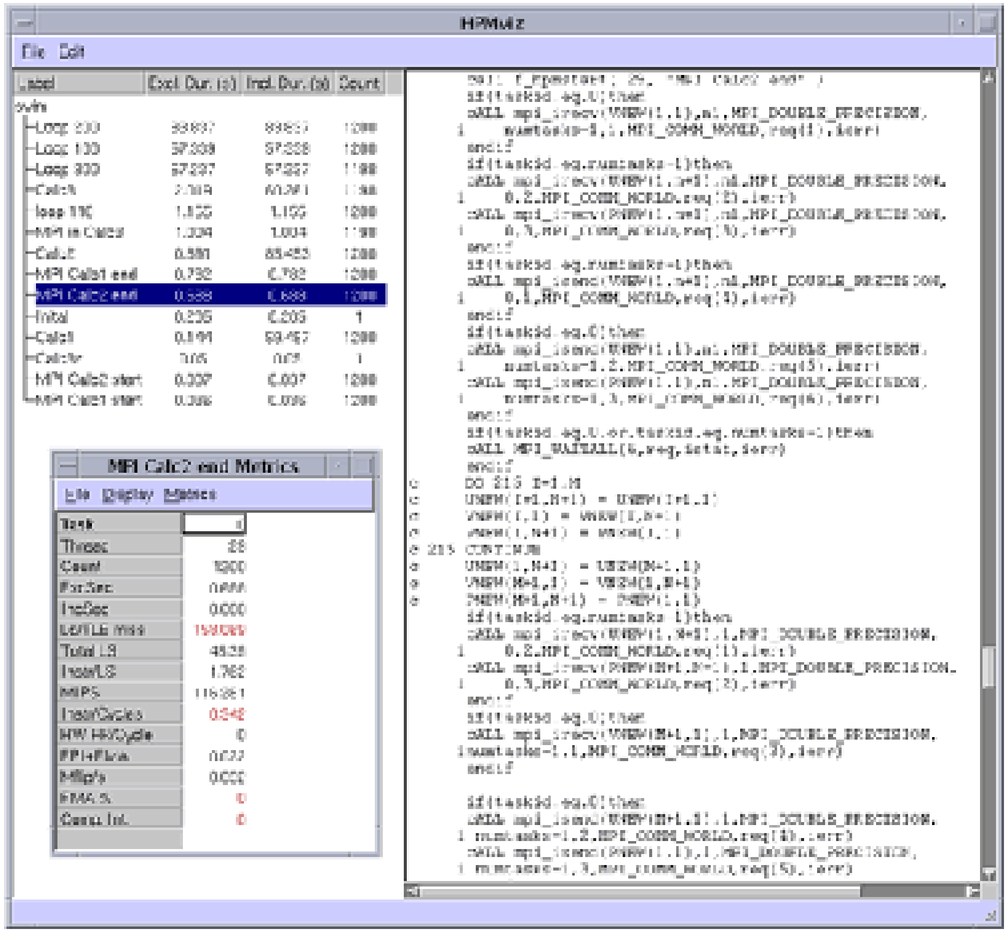
- Environment variables and XML input file dictate what gets measured

HPMToolkit from IBM ACTC

- Command line utility to gather aggregate counts.
  - PAPI version has been tested on IA32 & IA64
  - User can manually instrument code for more specific information
  - Reports derived metrics like SGI's perfex
- Libhpm for manual instrumentation
- Hpmviz is a GUI to view resulting data

http://www.ncsa.uiuc.edu/UserInfo/Resources/Software/Tools/HPMToolkit

hpmviz Screenshot



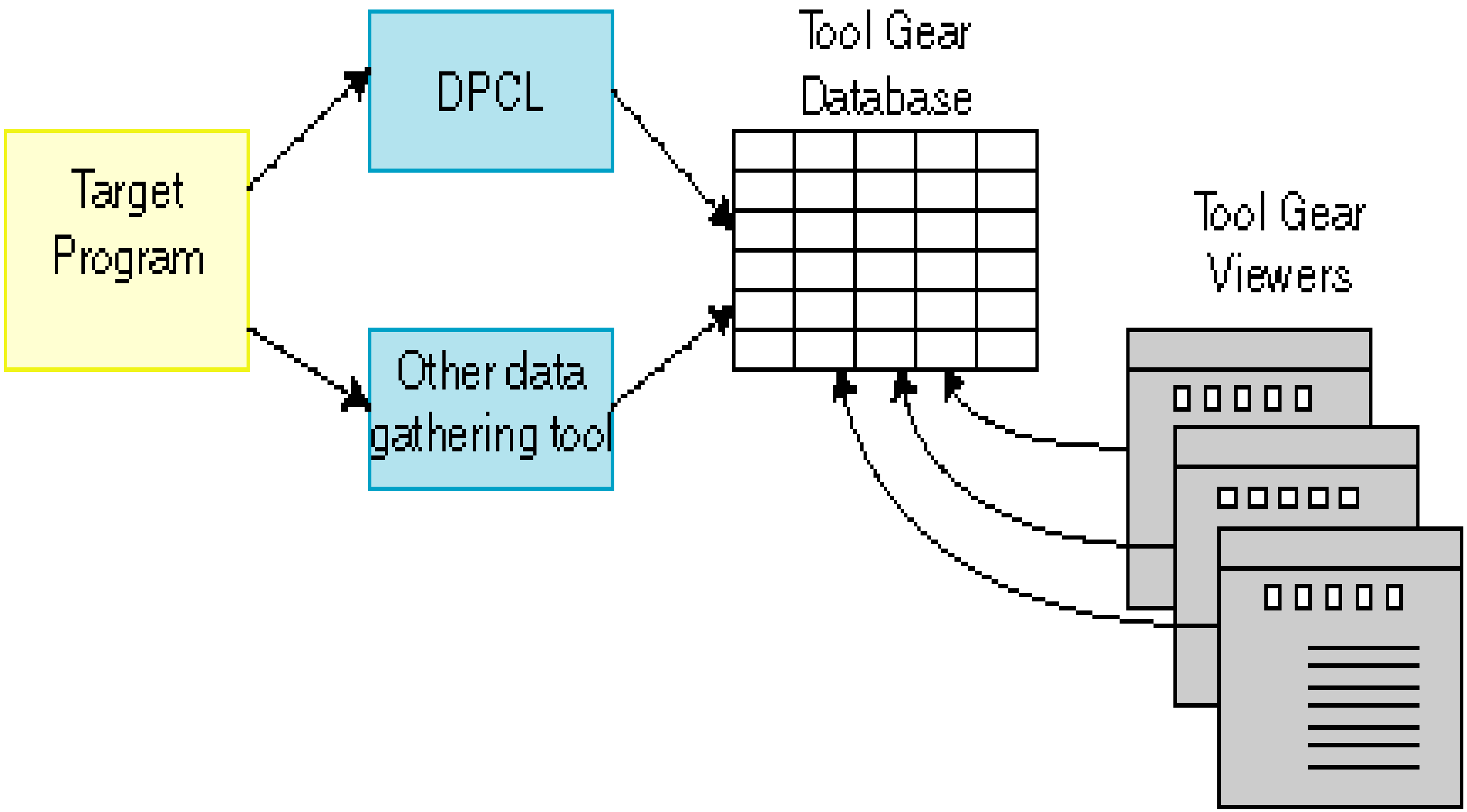
Libhpm Example

```
#include "libhpm.h"
hpmInit( tasked, "my program" );
hpmStart( 1, "outer call" );
do_work();
hpmStart( 2, "computing meaning of life" );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
```

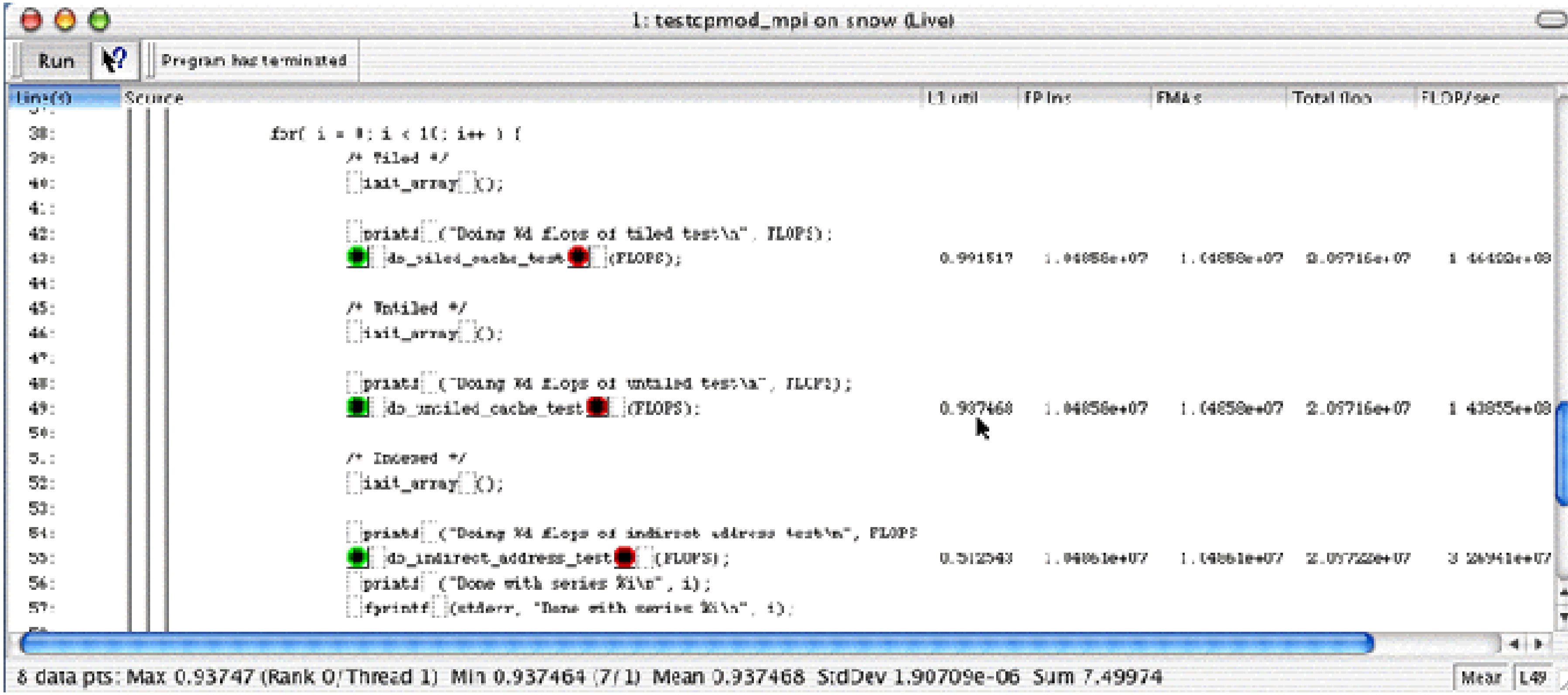
ToolGear Overview

- Dynamic instrumentation and analysis suite from LLNL
- Based on DPCL from IBM
  - Tested only on AIX
- Qt Front end can theoretically accept data from any source
- GUI displays instrumentable points
- Instrumented points update display with data in real time
- http://www.llnl.gov/CASC/tool_gear

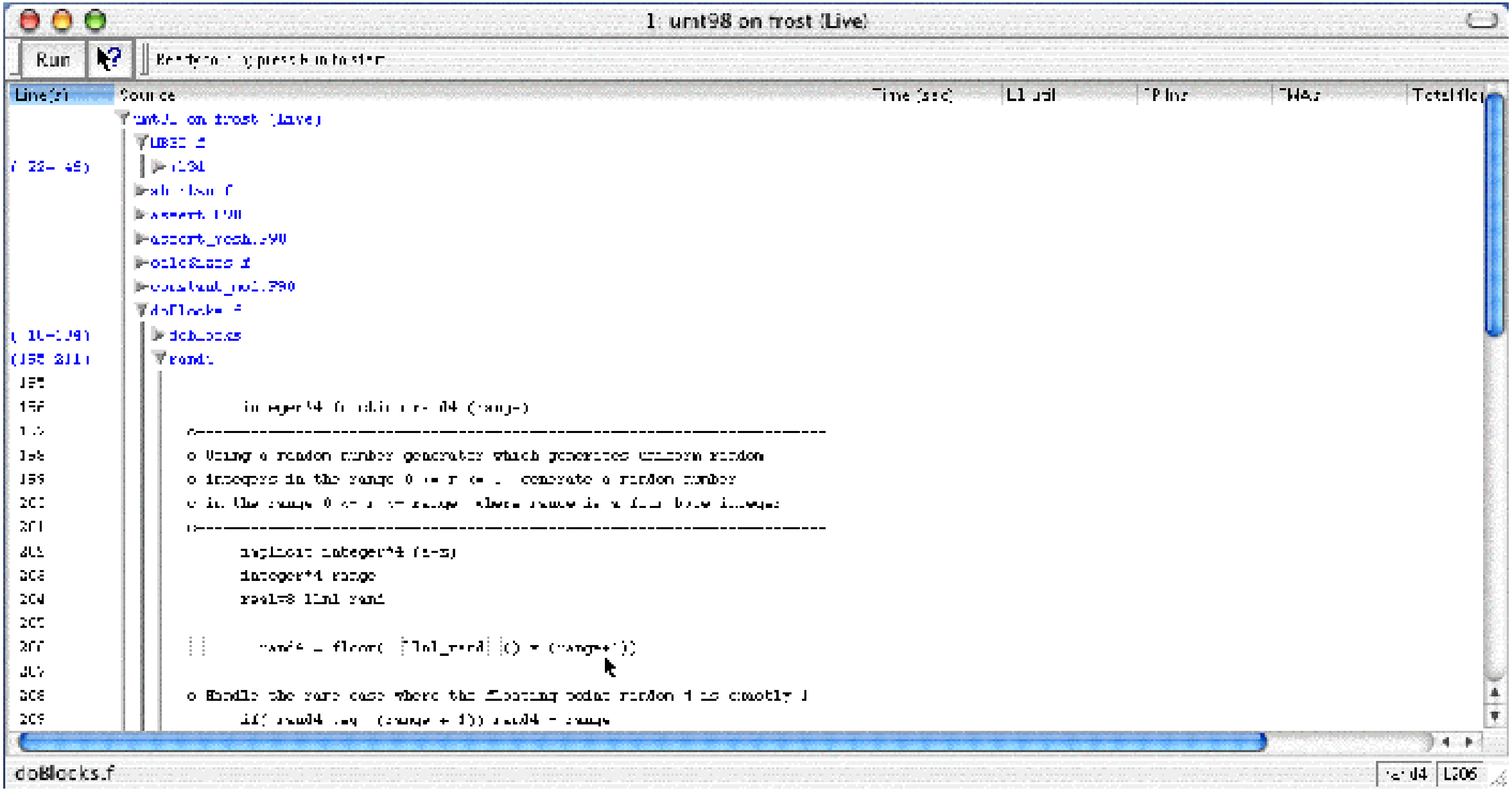
ToolGear Architecture



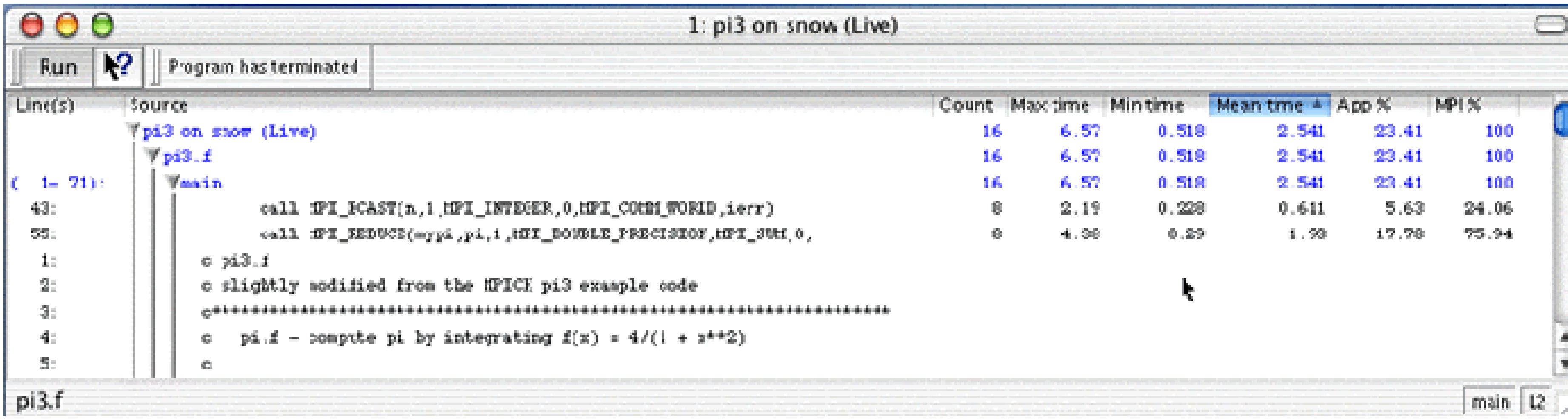
ToolGear Screenshot: Instrumentation



ToolGear Screenshot 2: Tree View



ToolGear Screenshot 3: MPI Profiling



DynaProf

- A portable tool to dynamically instrument serial and parallel programs for the purpose of performance analysis.
- Simple and intuitive command line interface like GDB.
- Java/Swing GUI.
- Instrumentation is done through the run-time insertion of function calls to specially developed performance probes.

Why the "Dyna" in DynaProf?

- Instrumentation:
  - Functions are contained in shared libraries.
  - Calls to those functions are generated at run-time.
  - Those calls are dynamically inserted into the program's address space.
- Built on DynInst and DPCL
- Can choose the mode of instrumentation, currently:
  - Function Entry/Exit
  - Call site Entry/Exit
  - One-shot

DynaProf Goals

- Make collection of run-time performance data easy by:
  - Avoiding instrumentation and recompilation
  - Avoiding perturbation of compiler optimizations
  - Providing complete language independence
  - Allowing multiple insert/remove instrumentation cycles

No source code required!

DynaProf Goals 2

– Using the same tool with different probes
– Providing useful and meaningful probe data
– Providing different kinds of probes
– Allowing custom probe development Make collection of run-time performance data easy by:

No source code required!

Dynaprof Probes

- perfometerprobe
  - Visualize hardware event rates in “real-time”
- papiprobe
  - Measure any combination of PAPI presets and native events
- wallclockprobe
  - Highly accurate elapsed wallclock time in microseconds.
- The latter 2 probes report:
  - Inclusive
  - Exclusive
  - 1 Level Call Tree

DynaProf Probe Design

- Probes export a few functions with loosely standardized interfaces.
- Easy to roll your own.
  - If you can code a timer, you can write a probe.
- DynaProf detects thread model.
- Probes dictate how the data is recorded and visualized.

Threads and Dynaprof Probes

- For threaded code, use the same probe!
- Dynaprof detects threads and loads a special version of the probe library.
- Each probe specifies what to do when a new thread is discovered.
- Each thread gets the same instrumentation.

PAPI Probe

- Can count any PAPI preset or Native event accessible through PAPI
- Can count multiple events
- Supports PAPI multiplexing
- Supports multithreading
  - AIX: SMP, OpenMP, Pthreads
  - Linux: SMP, OpenMP, Pthreads

Wallclock Probe

- Counts microseconds using RTC
- Supports multithreading
  - AIX: SMP, OpenMP, Pthreads
  - Linux: SMP, OpenMP, Pthreads

Reporting Probe Data

- The wallclock and PAPI probes produce very similar data.
- Both use a parsing script written in Perl.
  - wallclockrpt <file>
  - papiproberpt <file>
- Produce 3 profiles
  - Inclusive: $T_{function} = T_{self} + T_{children}$
  - Exclusive: $T_{function} = T_{self}$
  - 1-Level Call Tree: $T_{child} = \mathit{Inclusive}\ T_{function}$

Sample DynaProf Session

```
$./dynaprof
(dynaprof) load tests/swim
(dynaprof) list
DEFAULT_MODULE
swim.F
libm.so.6
libc.so.6
(dynaprof) list swim.F
MAIN__
inital_
calc1_
calc2_
calc3z_
calc3_
(dynaprof) list swim.F MAIN__
Entry
    Call s_wsle
    Call do_lio
    Call e_wsle
    Call s_wsle
    Call do_lio
    Call e_wsle
    Call calc3_
```

```
(dynaprof) use probes/papiprobe
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr module swim.F calc*
swim.F, inserted 4 instrumentation points
(dynaprof) run
papiprobe: output goes to
/home/mucci/dynaprof/tests/swim.1671
```

Instrumenting SWIM for IPC

```
(dynaprof) use probes/papiprobe PAPI_TOT_CYC, PAPI_TOT_INS
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr function swim.F calc*
Swim.F, inserted 3 instrumentation points
(dynaprof) instr
calc1_
calc2_
calc3_
calc3z_
```

Swim Benchmark: Cycles & Instructions

Exclusive Profile of Metric PAPI_TOT_INS.

Name	Percent	Total	Calls
TOTAL	100	1.723e+09	1
calc2	38.28	6.598e+08	120
calc1	32.31	5.567e+08	120
calc3	22.33	3.847e+08	118
unknown	7.084	1.221e+08	1

Inclusive Profile of Metric PAPI_TOT_INS.

Name	Percent	Total	SubCalls
TOTAL	100	1.723e+09	0
calc2	39.42	6.793e+08	1680
calc1	35.28	6.08e+08	1800
calc3	22.87	3.942e+08	1652

1-Level Inclusive Call Tree of Metric PAPI_TOT_INS.

Parent/-Child	Percent	Total	Calls
TOTAL	100	1.723e+09	1
calc1	100	6.08e+08	120
- fsav	0.02065	1.255e+05	120
- mpi_irecv	0.03132	1.904e+05	120
- mpi_isend	0.05911	3.593e+05	120
- mpi_isend	0.06434	3.912e+05	120
-mpi_waitall	0.9013	5.479e+06	120
- mpi_irecv	0.03132	1.904e+05	120
- mpi_irecv	0.03132	1.904e+05	120
- mpi_isend	0.05356	3.256e+05	120
- mpi_isend	0.05079	3.088e+05	120
-mpi_waitall	6.813	4.142e+07	120
- mpi_irecv	0.03132	1.904e+05	120
- mpi_irecv	0.03132	1.904e+05	120
- mpi_isend	0.07504	4.562e+05	120
- mpi_isend	0.06757	4.108e+05	120
-mpi_waitall	0.161	9.791e+05	120
calc2	100	6.793e+08	120

Exclusive Profile of Metric PAPI_TOT_CYC.

Name	Percent	Total	Calls
TOTAL	100	3.181e+09	1
calc2	34.85	1.108e+09	120
calc1	33.48	1.065e+09	120
calc3	26.1	8.301e+08	118
unknown	5.568	1.771e+08	1

Inclusive Profile of Metric PAPI_TOT_CYC.

Name	Percent	Total	SubCalls
TOTAL	100	3.181e+09	0
calc2	35.98	1.144e+09	1680
calc1	35.61	1.133e+09	1800
calc3	26.88	8.55e+08	1652

1-Level Inclusive Call Tree of Metric PAPI_TOT_CYC.

Parent/-Child	Percent	Total	Calls
TOTAL	100	3.181e+09	1
calc1	100	1.133e+09	120
- fsav	0.03432	3.887e+05	120
- mpi_irecv	0.07356	8.332e+05	120
- mpi_isend	0.0663	7.51e+05	120
- mpi_isend	0.0739	8.371e+05	120
-mpi_waitall	0.7189	8.143e+06	120
- mpi_irecv	0.1646	1.864e+06	120
- mpi_irecv	0.03407	3.859e+05	120
- mpi_isend	0.1867	2.115e+06	120
- mpi_isend	0.06067	6.872e+05	120
-mpi_waitall	4.22	4.78e+07	120
- mpi_irecv	0.03979	4.506e+05	120
- mpi_irecv	0.03008	3.407e+05	120
- mpi_isend	0.1014	1.148e+06	120
- mpi_isend	0.07568	8.573e+05	120
-mpi_waitall	0.1076	1.219e+06	120
calc2	100	1.144e+09	120

Swim Benchmark: Instructions per Cycle

Exclusive Profile of Metric PAPI_TOT_INS.

Name	Percent	Total	Calls
TOTAL	100	1.723e+09	1
calc2	38.28	6.598e+08	120
calc1	32.31	5.567e+08	120
calc3	22.33	3.847e+08	118
unknown	7.084	[illegible]	1

Inclusive Profile of Metric PAPI_TOT_INS.

Name	Percent	Total	SubCalls
TOTAL	100	1.723e+09	0
calc2	39.42	6.793e+08	1680
calc1	35.28	6.08e+08	1800
calc3	22.87	3.942e+08	[illegible]

1-Level Inclusive Call Tree of Metric PAP[illegible]

Parent/-Child	Percent	Total	[illegible]
TOTAL	100	[illegible]	[illegible]
calc1	100	[illegible]	[illegible]
- fsav	0.02065	[illegible]	[illegible]
- mpi_irecv	0.03132	[illegible]	[illegible]
- mpi_isend	0.05911	[illegible]	[illegible]
- mpi_isend	0.06434	[illegible]	[illegible]
-mpi_waitall	0.9013	[illegible]	[illegible]
- mpi_irecv	0.03132	[illegible]	[illegible]
- mpi_irecv	0.03132	[illegible]	[illegible]
- mpi_isend	0.05356	[illegible]	[illegible]
- mpi_isend	0.05079	[illegible]	[illegible]
-mpi_waitall	6.813	[illegible]	[illegible]
- mpi_irecv	0.03132	[illegible]	[illegible]
- mpi_irecv	0.03132	[illegible]	[illegible]
- mpi_isend	0.07504	4.562e+05	[illegible]
- mpi_isend	0.06757	4.108e+05	[illegible]
-mpi_waitall	0.161	9.791e+05	[illegible]

Exclusive Profile of Metric PAPI_TOT_CYC.

Name	Percent	Total	Calls
TOTAL	100	3.181e+09	1
calc2	34.85	1.108e+09	120
calc1	33.48	1.065e+09	120
calc3	26.1	8.301e+08	118
unknown	5.568	[illegible]	1

Inclusive Profile of Metric PAPI_TOT_CYC.

Name	Percent	Total	SubCalls
TOTAL	100	3.181e+09	0
calc2	35.98	1.144e+09	1680
calc1	35.61	1.133e+09	1800
[illegible]	26.88	8.55e+08	1652

[illegible]ive Call Tree of Metric PAPI_TOT_CYC.

[illegible]	[illegible]	Total	Calls
[illegible]	[illegible]	3.181e+09	1
[illegible]	[illegible]	1.133e+09	120
[illegible]	[illegible]	3.887e+05	120
[illegible]	[illegible]	8.332e+05	120
[illegible]	[illegible]	7.51e+05	120
[illegible]	[illegible]	8.371e+05	120
[illegible]	[illegible]	8.143e+06	120
[illegible]	[illegible]	1.864e+06	120
[illegible]	[illegible]	3.859e+05	120
[illegible]	[illegible]	2.115e+06	120
[illegible]	[illegible]	6.872e+05	120
[illegible]	[illegible]	4.78e+07	120
[illegible]	[illegible]	4.506e+05	120
[illegible]	0.03008	3.407e+05	120
[illegible]	0.1014	1.148e+06	120
[illegible]	0.07568	8.573e+05	120
[illegible]	0.1076	1.219e+06	120

calc2	0.59
calc1	0.53
calc3	0.46

DynaProf GUI

- Displays module tree for instrumentation
- Simple selection of probes and instrumentation points
- Single-click execution of common DynaProf commands
- Coupling of probes and visualizers (e.g. Perfometer)

Dynaprof Status

- It's a bit rough
- Supported Platforms
  - Using DynInst 3.0
    - Linux 2.x
    - AIX 4.3/5
  - Using DPCL (formal MPI support)
    - AIX 4.3
    - AIX 5
- Includes:
  - Java/Swing GUI
  - User's Guide
  - Probe libraries

- Available as a development snapshot from:

http://www.cs.utk.edu/~mucci/dynaprof

The End

Thanks!

Philip J. Mucci
LBNL 50B-3207
510 486-8616
pjmucci@lbl.gov
mucci@cs.utk.edu