PAPI and Hardware Performance Analysis Tools

Los Alamos National Laboratory Visit, February 28, 2003

Philip Mucci, Research Consultant
Innovative Computing Laboratory/UTK
Performance Evalutation Research Center/LBNL

mucci@cs.utk.edu
http://icl.cs.utk.edu/projects/papi

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

- PAPI
- Performance Monitoring Hardware
- Performance Analysis Tools
  - Dynaprof
  - Other tools
- Trends

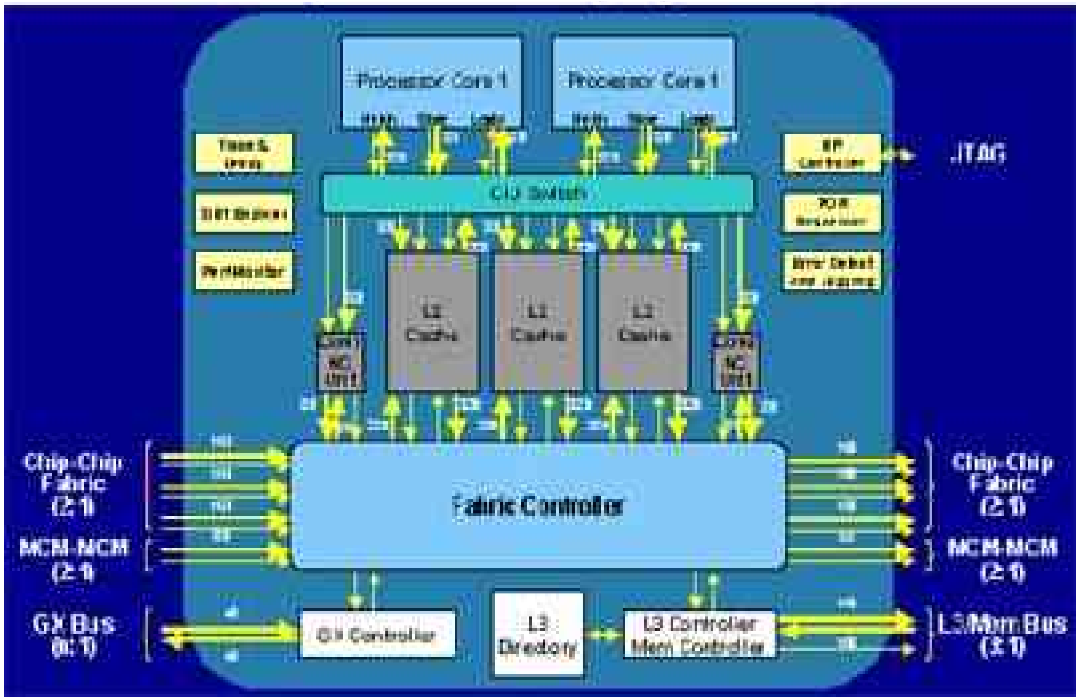
PAPI Goal

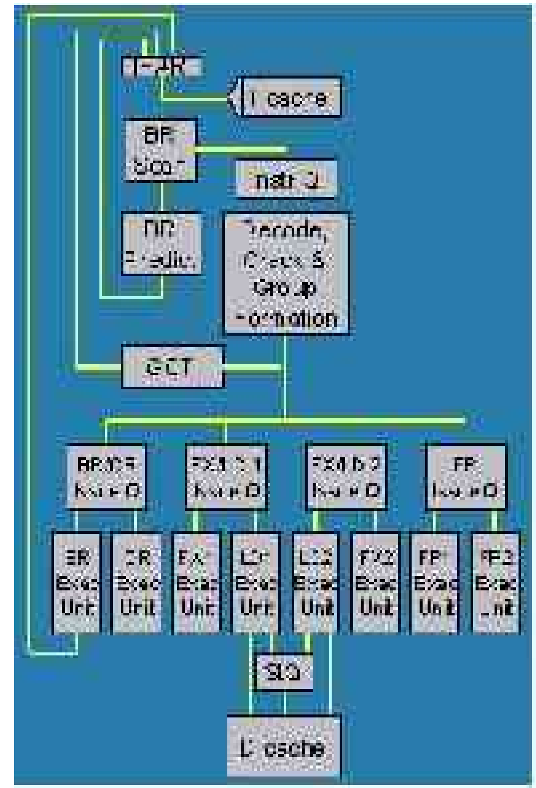
- To understand why the application performs as it does.
  - Optimize the application's performance.
  - Evaluate the algorithms efficiency.
  - Generate an application signature.
  - Develop a performance model.
- Data is NOT PORTABLE, but the interface is...

Overview of Hardware Counters

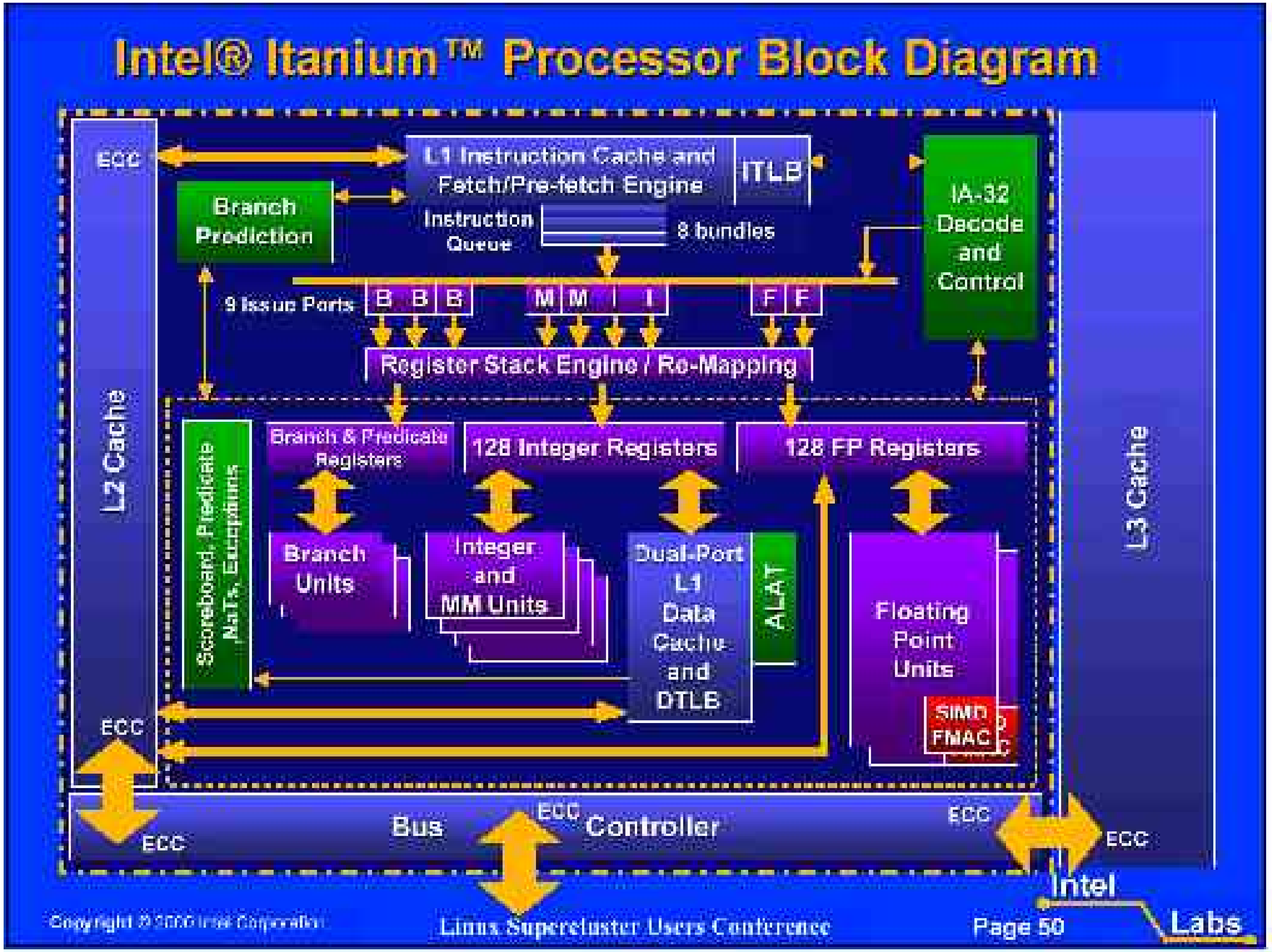
- Small number of registers dedicated for performance monitoring functions.
  - AMD Athlon, 4 counters
  - Pentium <= III, 2 counters
  - Pentium IV, 18 counters
  - IA64, 4 counters
  - Alpha 21x64, 2 counters
  - Power 3, 8 counters
  - Power 4, 6 counters
  - UltraSparc II, 2 counters
  - MIPS R14K, 2 counters

Power 4 Diagram



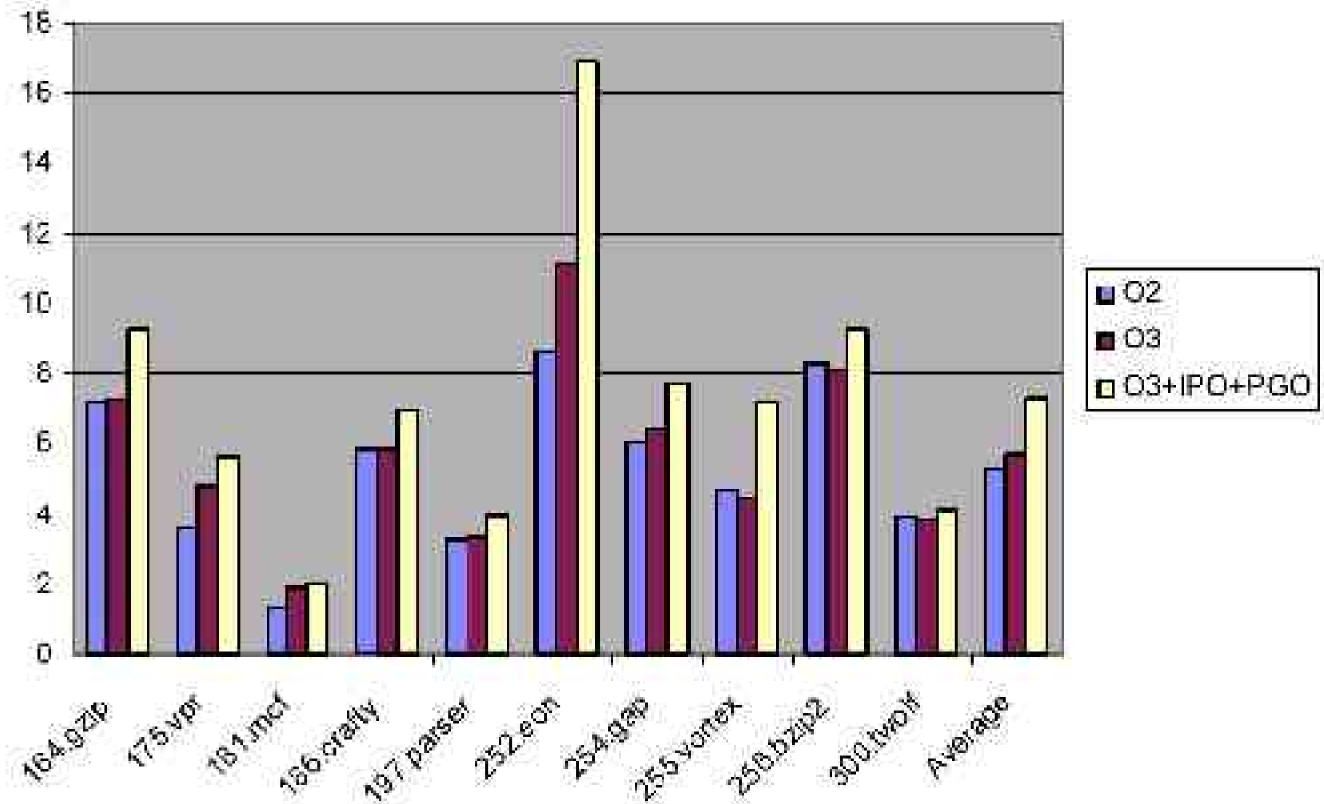


IA64 Block Diagram

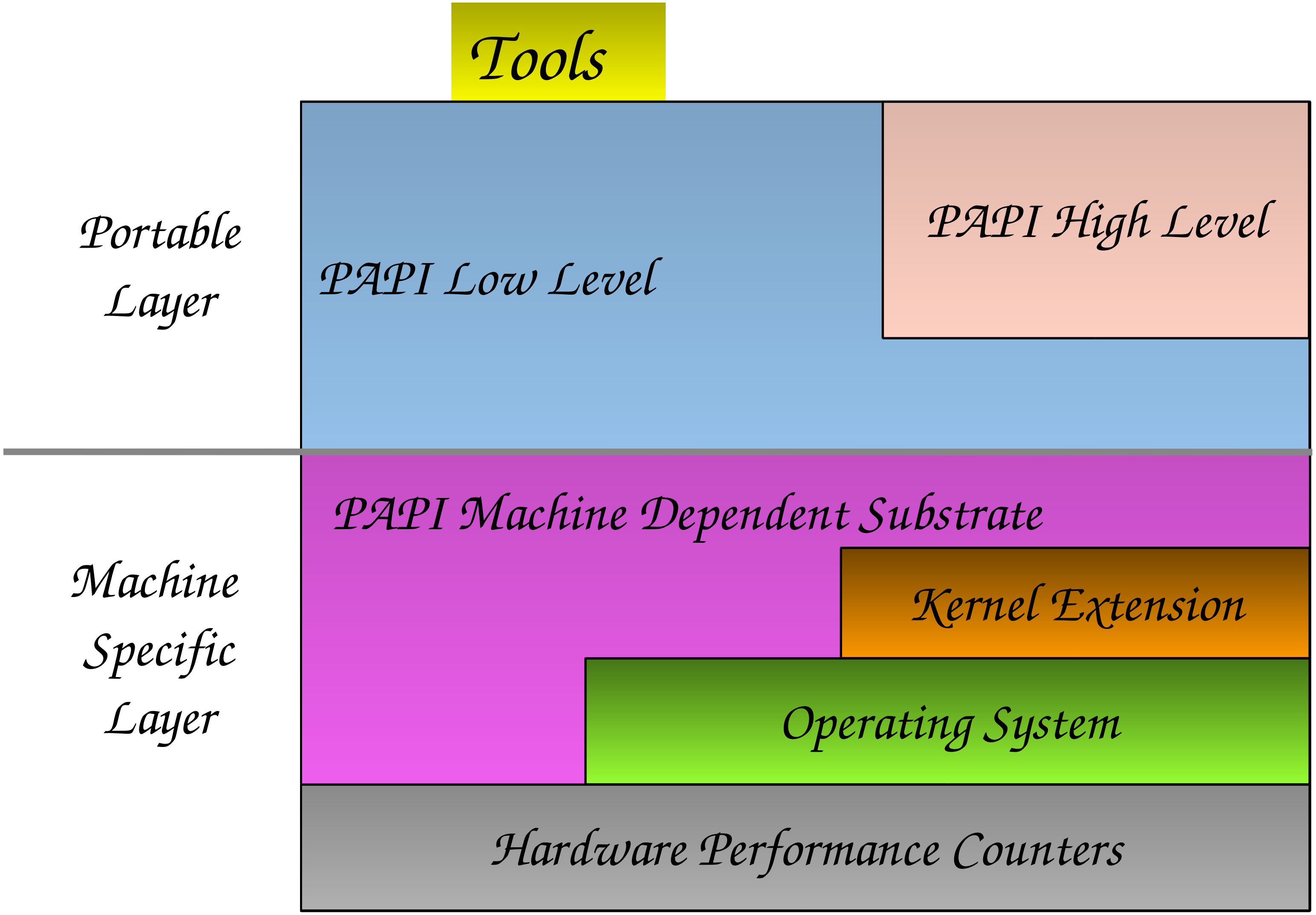


Importance of Optimization

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

```
> tests/avail
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              Avail    Deriv    Description (Note)
PAPI_L1_DCM       0x80000000        Yes      No       Level 1 data cache misses
PAPI_L1_ICM       0x80000001        Yes      No       Level 1 instruction cache
    misses
PAPI_L2_DCM       0x80000002        No       No       Level 2 data cache misses
PAPI_L2_ICM       0x80000003        No       No       Level 2 instruction cache
    misses
PAPI_L3_DCM       0x80000004        No       No       Level 3 data cache misses
PAPI_L3_ICM       0x80000005        No       No       Level 3 instruction cache
    misses
PAPI_L1_TCM       0x80000006        Yes      Yes      Level 1 cache misses
PAPI_L2_TCM       0x80000007        Yes      No       Level 2 cache misses
PAPI_L3_TCM       0x80000008        No       No       Level 3 cache misses
PAPI_CA_SNP       0x80000009        No       No       Requests for a snoop
PAPI_CA_SHR       0x8000000a        No       No       Requests for shared
PAPI_CA_CLN       0x8000000b        No       No       Requests for clean c
PAPI_CA_INV       0x8000000c        No       No       Requests for cache l
.
.
```

http://icl.cs.utk.edu/projects/papi/files/html_man/papi_presets.html

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 IRIX calls
- Requires no setup code
- Restrictions:
  - Allows only PAPI presets
  - Not thread safe
  - Only aggregate counters

High-level API Calls

- **`PAPI_num_counters()`**
  - Returns the number of available counters
- **`PAPI_start_counters(int *cntrs, int alen)`**
  - Start counters
- **`PAPI_stop_counters(long_long *vals, int alen)`**
  - Stop counters and put counter values in array
- **`PAPI_accum_counters(long_long *vals, int alen)`**
  - Accumulate counters into array and reset
- **`PAPI_read_counters(long_long *vals, int alen)`**
  - Copy counter values into array and reset counters
- **`PAPI_flops(float *rtime, float *ptime, long_long *flpins, float *mflops)`**
  - Wallclock time, process time, FP ins since start,
  - Mflop/s since last call

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 (SMP, OpenMP, Pthreads)
- 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.
  - Implementation was based on MPX done by…… John May!

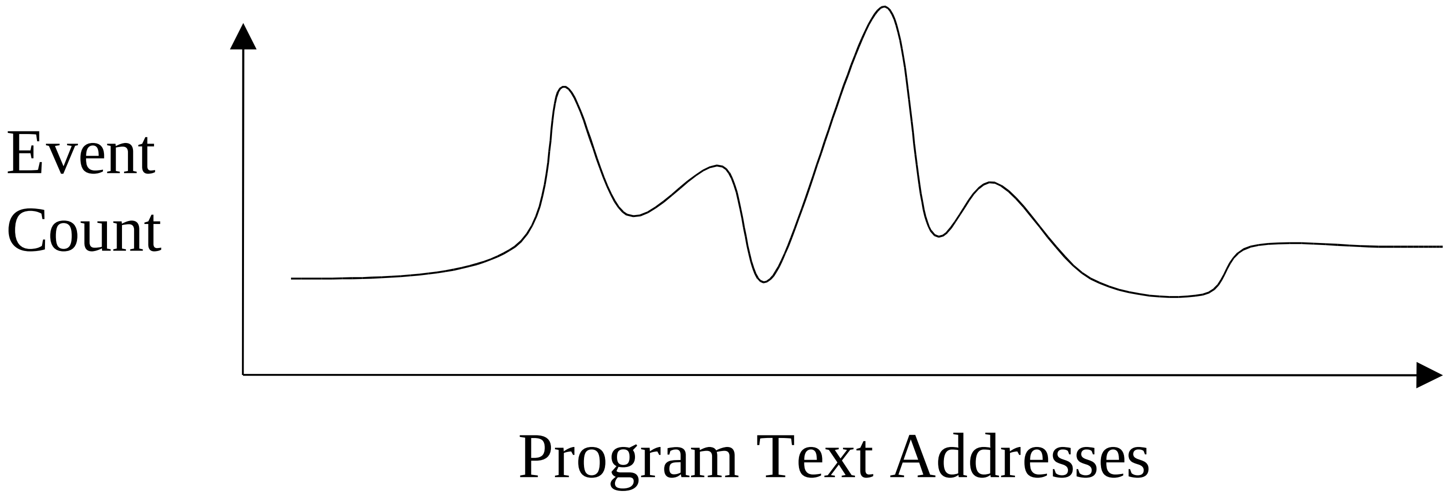
Interrupts on Counter Overflow

- 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 2.3.3 Release

- Additional Platforms
  - IBM PPC604, 604e, Power 3
  - Intel x86
  - Sun UltraSparc I/II/III
  - SGI MIPS R10K/R12K/R14K
  - Compaq Alpha 21164/21264 with DADD/DCPI
  - Itanium
  - Itanium 2
  - Power 4
  - AIX 5, Power 3, 604e
- Enhancements
  - Static/dynamic memory info
  - IA64 hardware profiling
  - Misc bug fixes
- Sample Tools
  - Perfometer
  - Trapper
  - Dynaprof

PAPI 3.0

- Using lessons learned from years earlier
  - Substrate code: 90% used only 10% of the time
  - In practice, it was never used
- Redesign for:
  - Robustness
  - Feature set
  - Simplicity
  - Portability to new platforms

Some PAPI 3.0 Features

- Multiway multiplexing
  - Use all available counter registers instead of one per time slice. (Just 1 additional register means 2x increase in accuracy)
- Superb performance
  - Pentium 4, a PAPI_read() costs 230 cycles.
  - Register access alone costs 100 cycles.
- Programmable events
  - Event Thresholding
  - Instruction matching
  - Per event counting domains

PAPI 3.0 Features 2

- Third-party interface
  - Allows PAPI to control counters in other threads of execution
- Internal timer/signal/thread abstractions
- Additional internal layered API to support robust extensions
- Advanced profiling functions for Event Address Sampling. (branch, cache, etc...)

PAPI 3.0 Features 3

- System-wide counting
- High level API made thread safe
- New language bindings
  - Java
  - Lisp
  - Matlab
- New tools to be included: papirun and papiprof from Rice.

PAPI 3.0 Release Targets

- Release expected May, 2003
- Additional platforms
  - Cray X-1
  - AMD Opteron/K8
  - Nec SX-6?
  - Blue Gene (BG/L)?

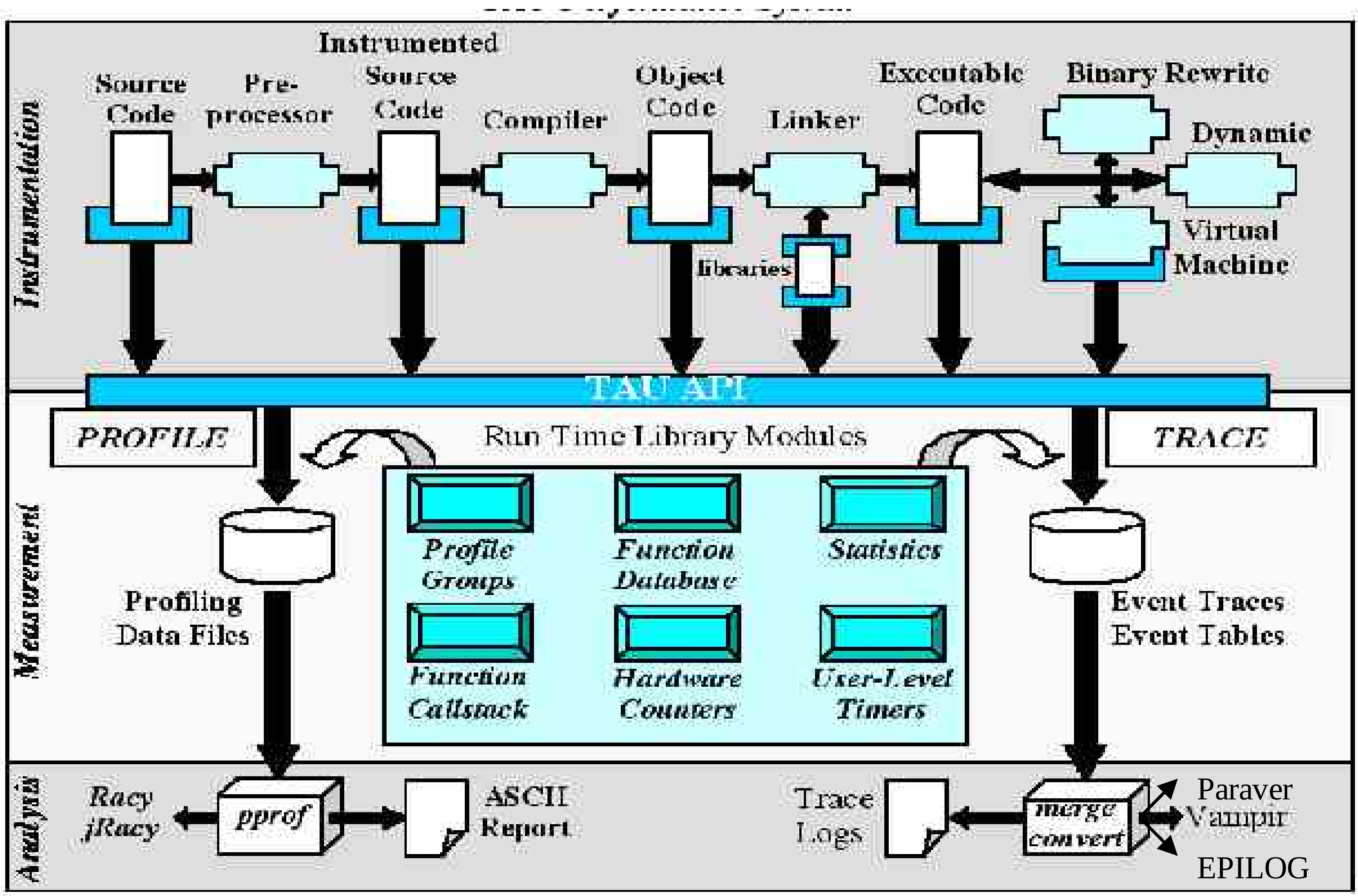
Paradyn from U. Wisconsin

- From Barton Miller's Group
- DynInst based dynamic discovery of bottlenecks
- Different visualization plugins
- Supports all forms of parallelism
- New version will do discovery based on hardware metrics
  - Memory stall time

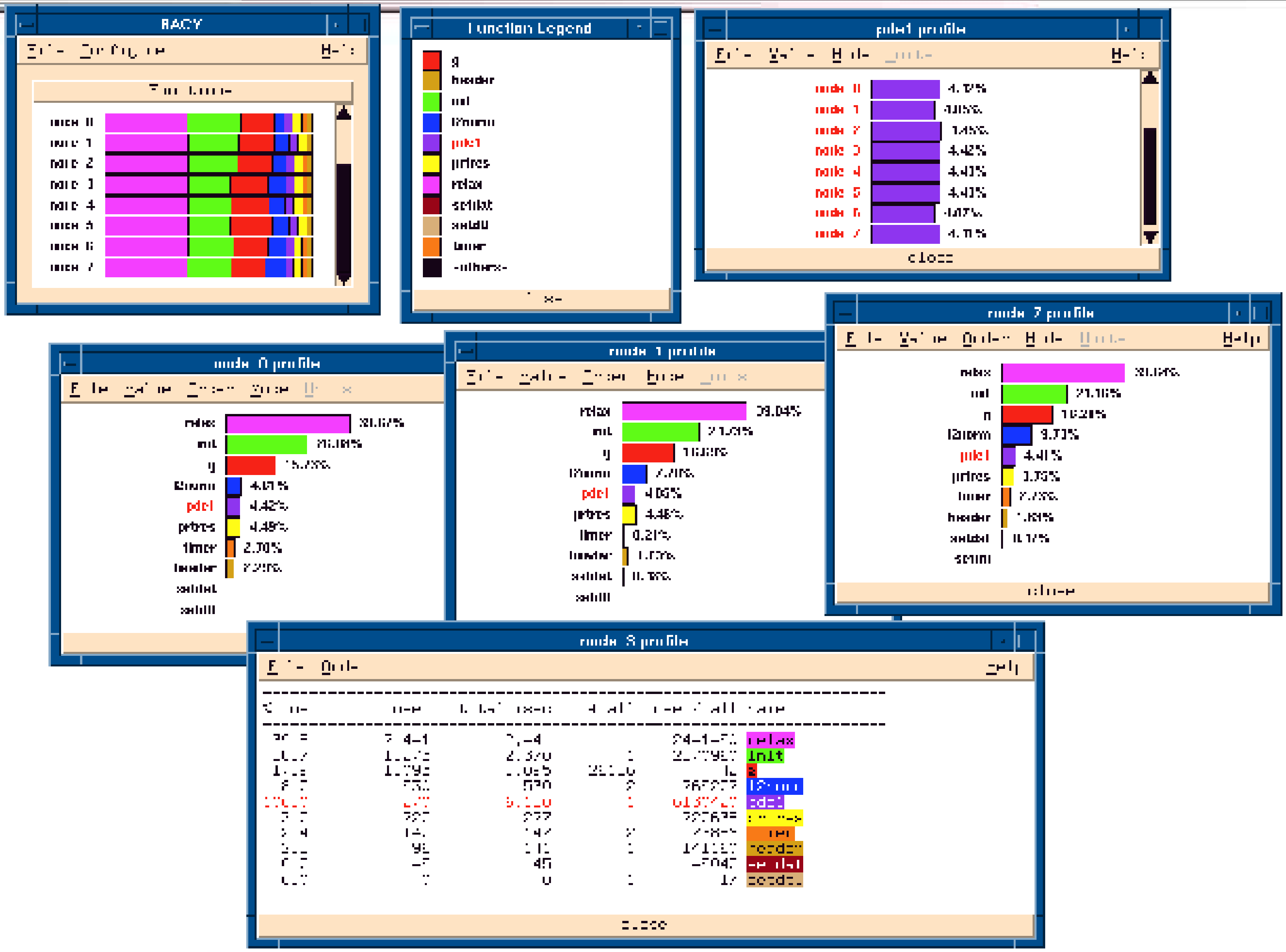
Tuning and Analysis Utilities

- From Allen Maloney's Group at U. Oregon
- Source or DynInst based
- Different visualization plugins
- Supports all forms of parallelism
- Integration with Vampir

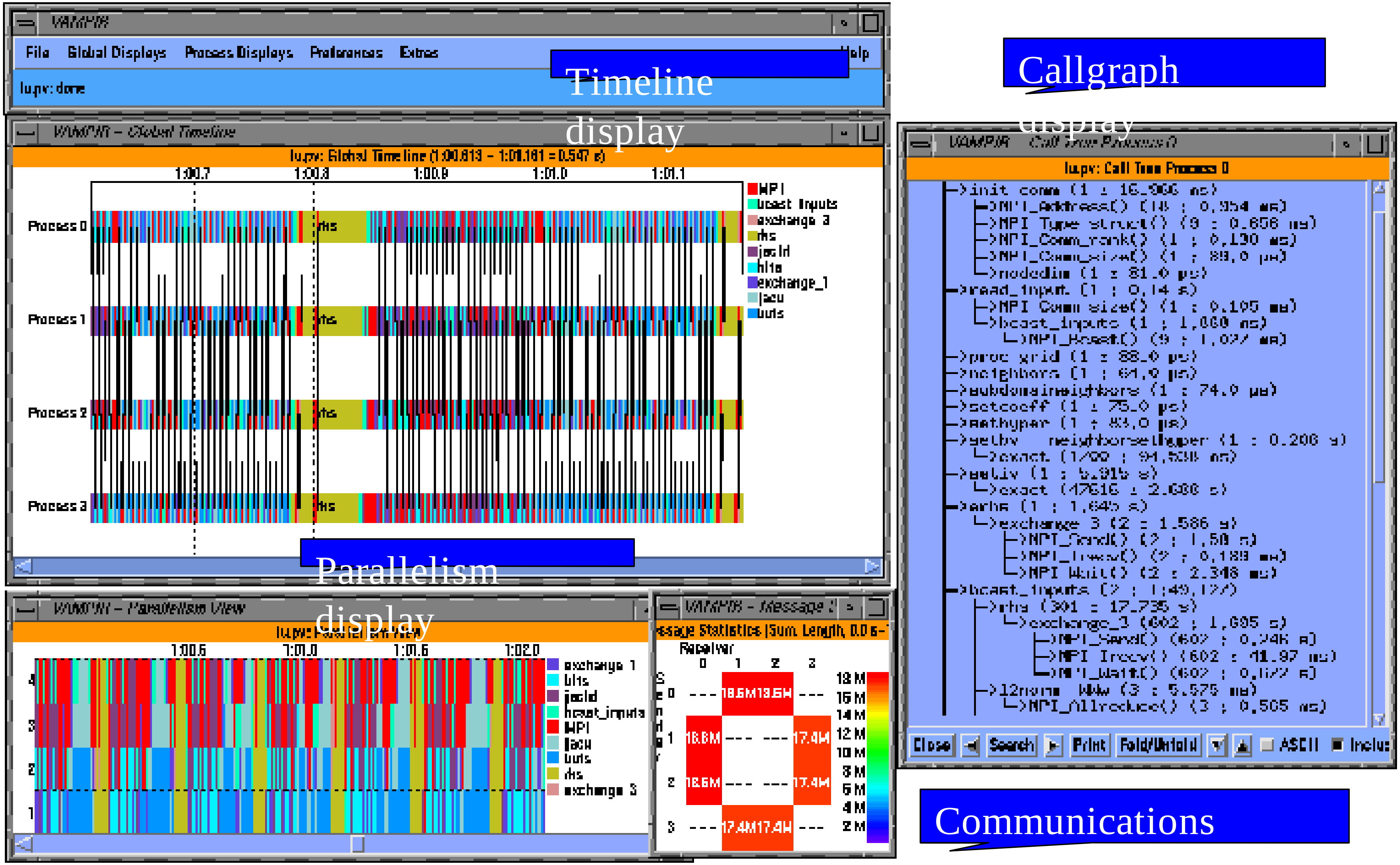
TAU Performance System Architecture



TAU Screenshot

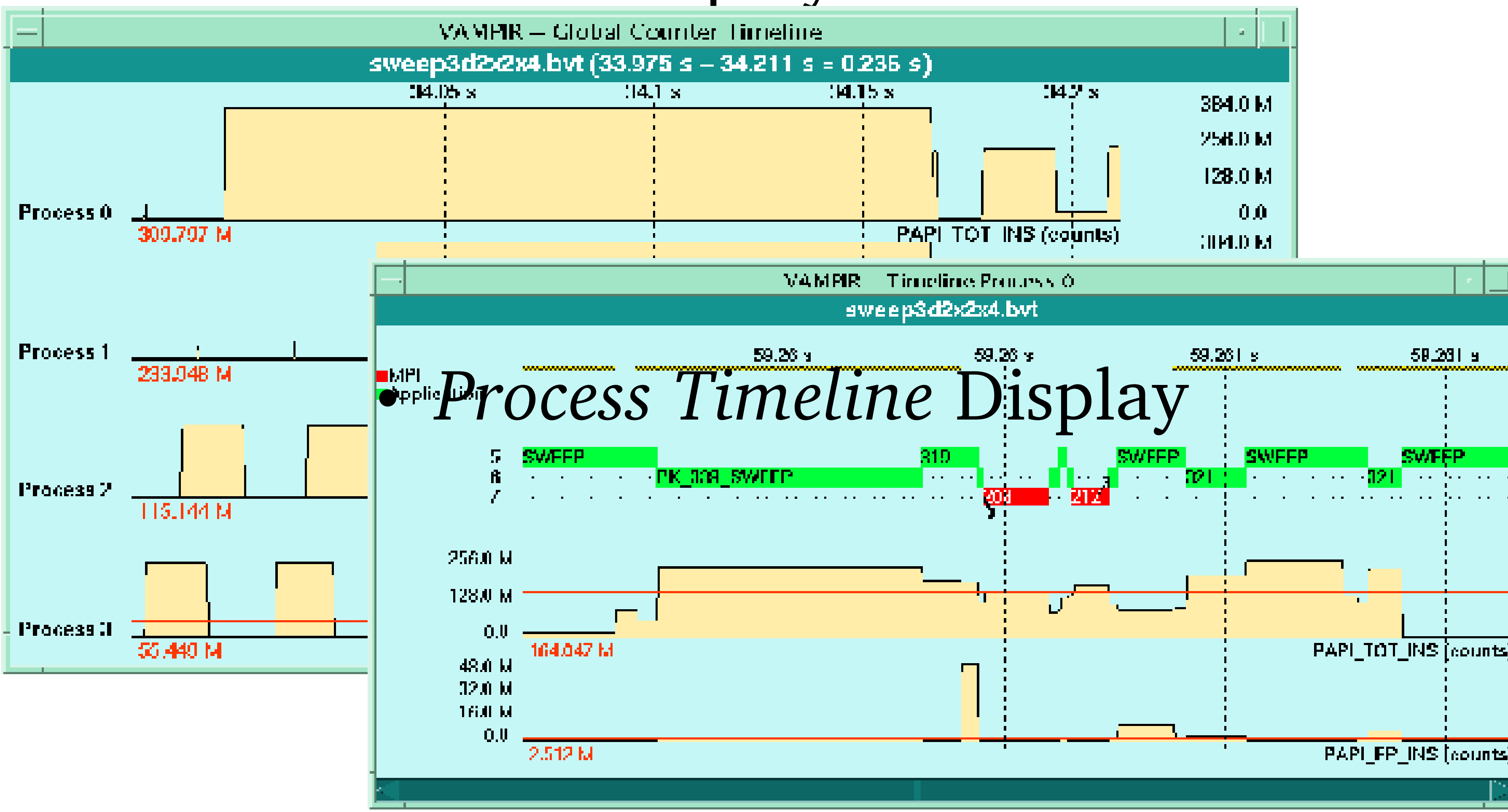


Vampir (NAS Parallel Benchmark – LU)



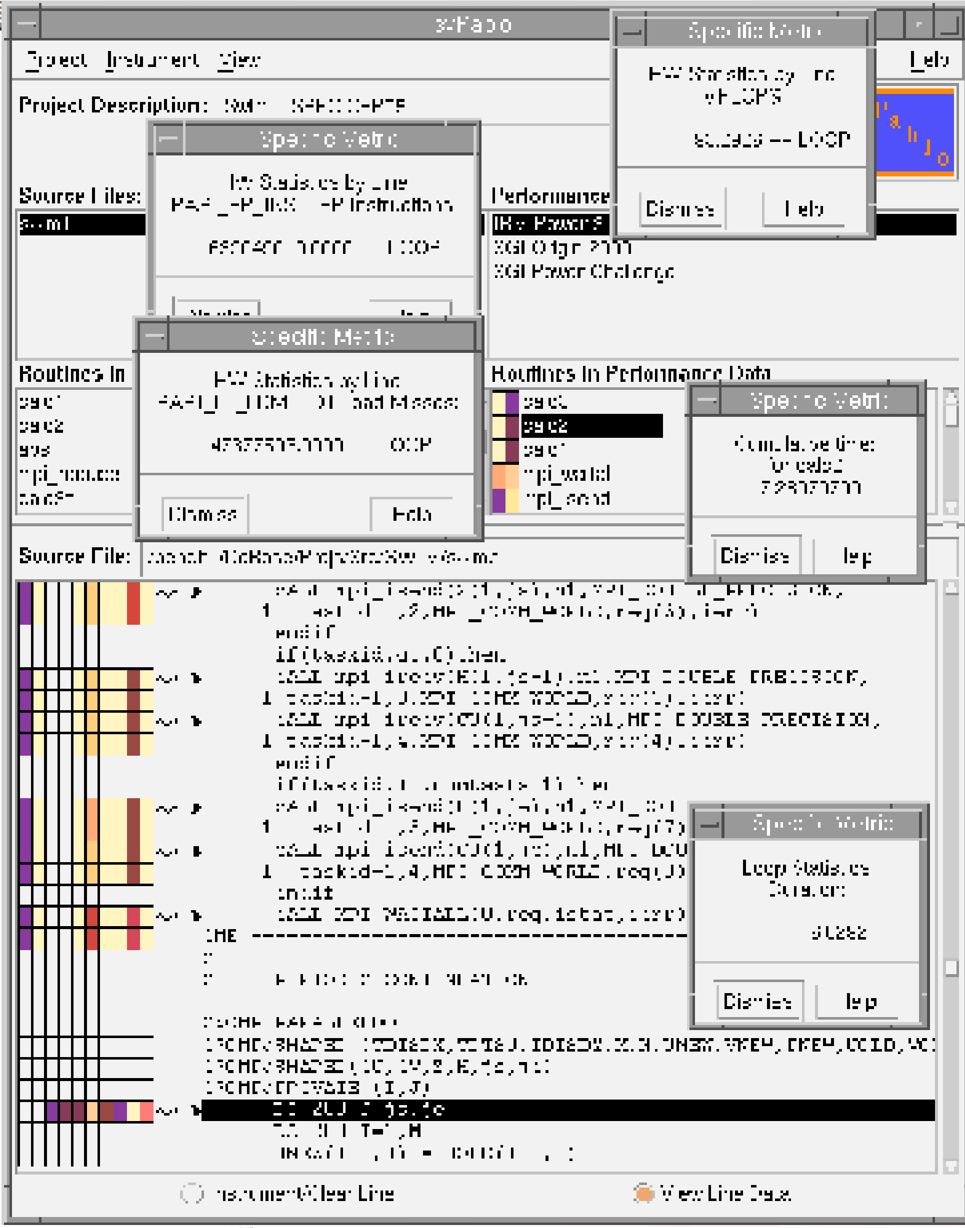
Vampir v3.x: HPM Counter

- *Counter Timeline* Display



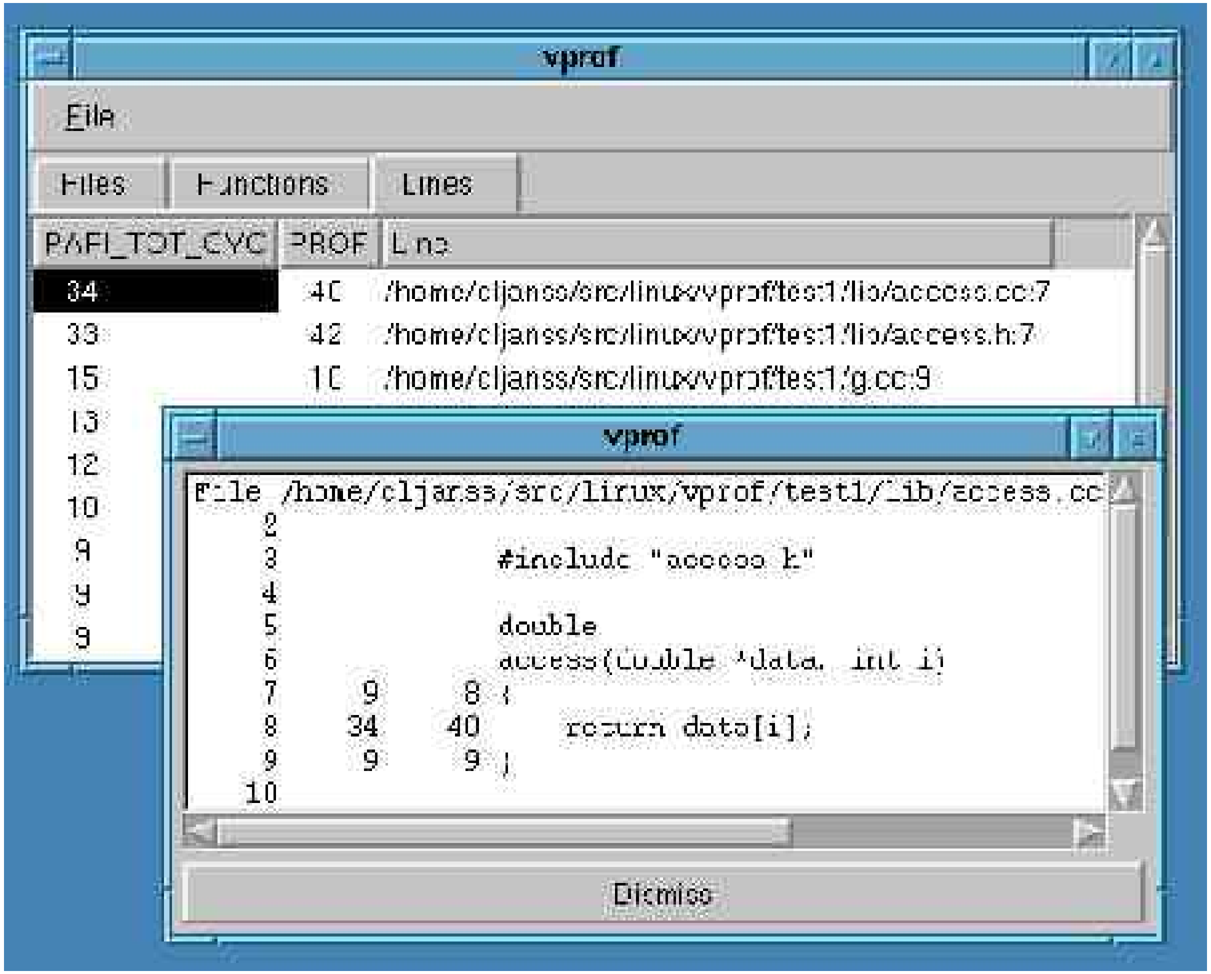
- *Process Timeline* Display

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

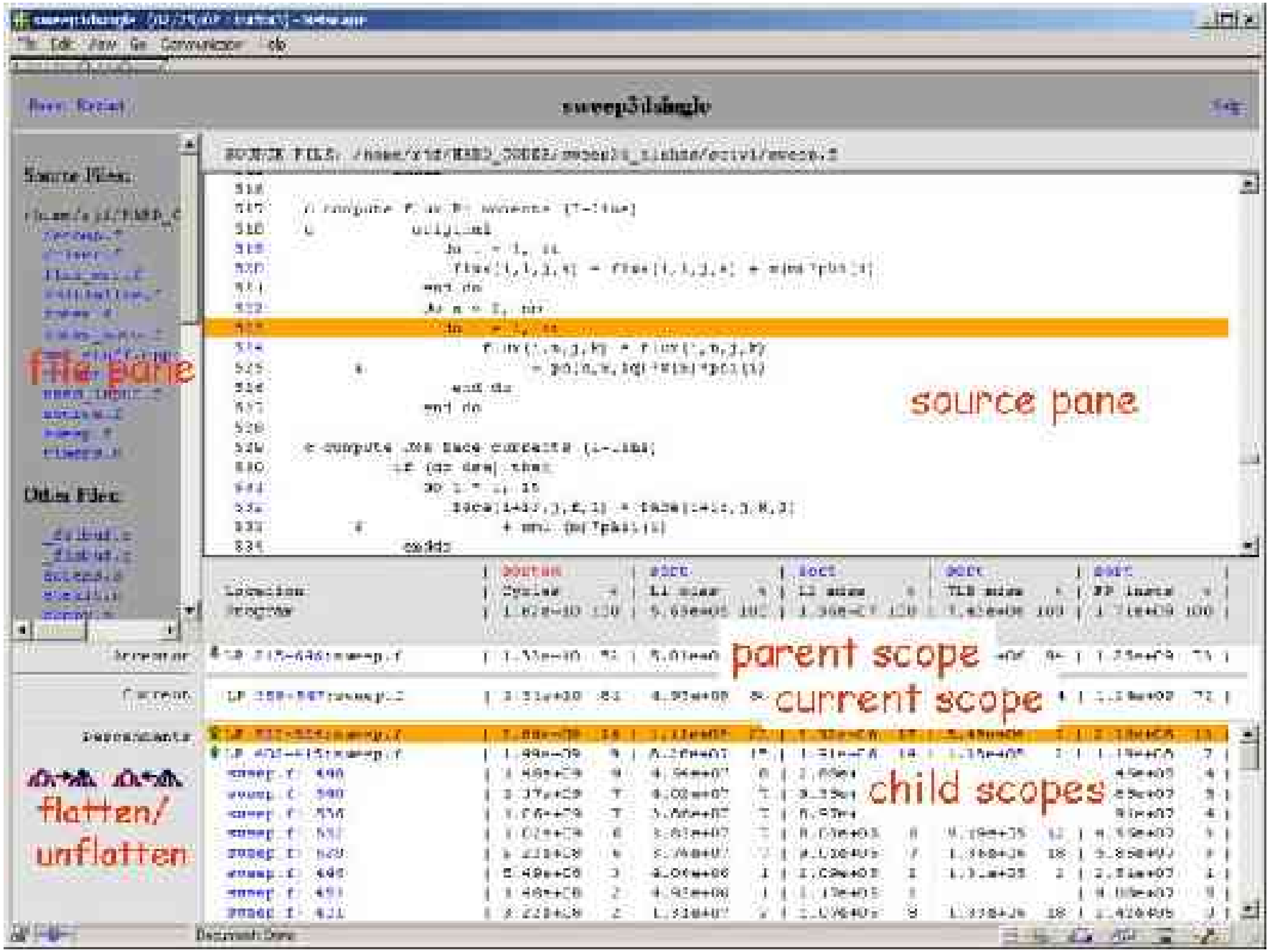


- Based on statistical sampling of the hardware counters
- Must instrument the source
- Ported to other architectures for generalized use
- Serial codes, parallel with modification
- Freely available

HPCTools from Rice University

- Uses PC-sampling on event thresholds
- Combines numerous modules
  - Collection: papirun, equivalent to "ssrun"
  - Binary loop/CFG recovery: bloop
  - 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



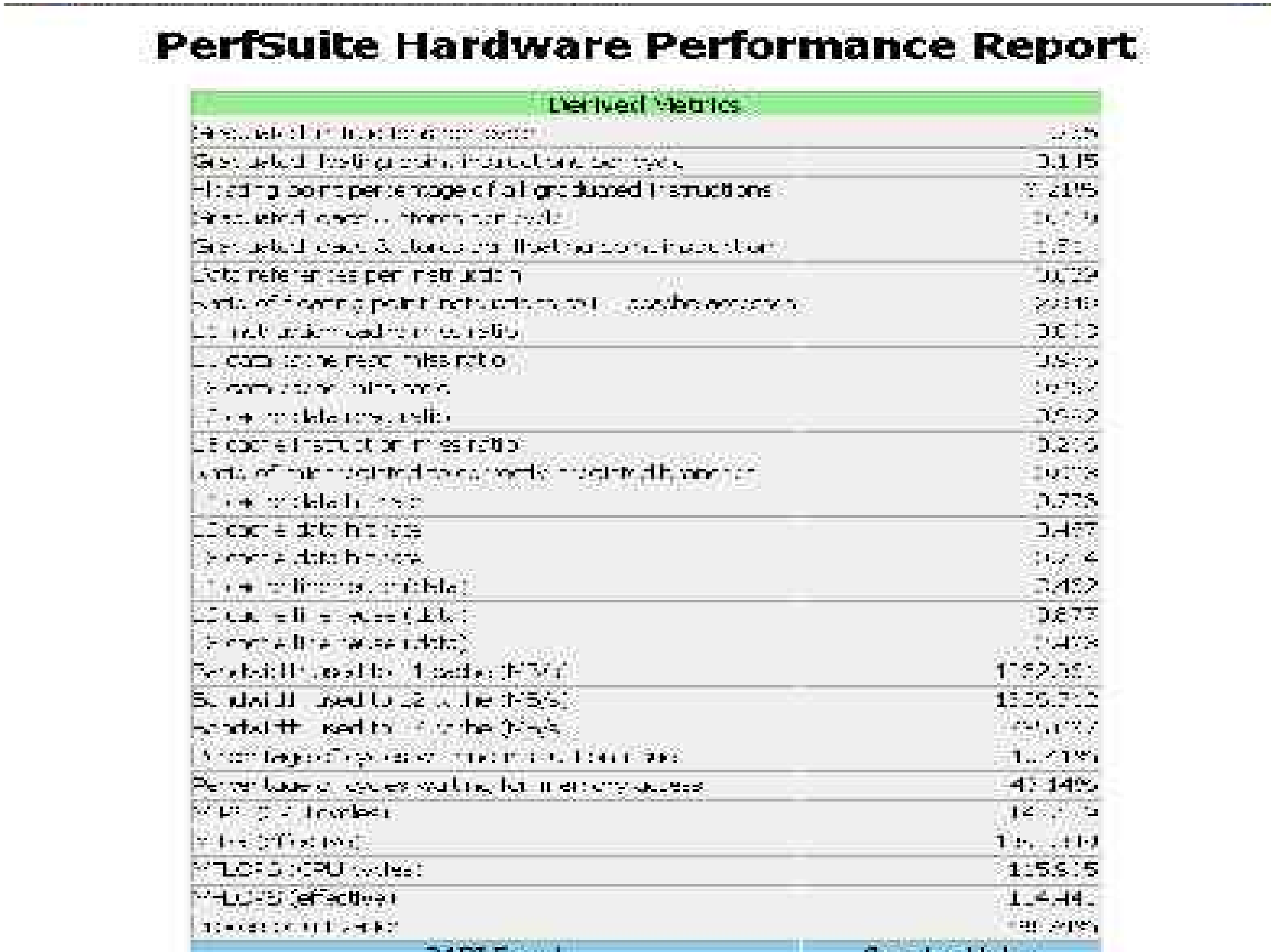
Trapper from PAPI

- Tool that allows the user to write functions that get executed at:
  - Process Creation/Deletion
  - Thread Creation/Deletion
- Actually, any function can be "preempted".
- The object code of the application isn't modified.
- Works by "preloading" special shared libraries and overloading function calls in cooperation with the run-time linker.

psrun/psprocess from NCSA

- Gives aggregate counts and derived metrics for unmodified source code.
- 2 phase
  - psrun: Collection, output to XML
  - psprocess: Formatting, output via XML/XSL parser
- Based on
  - PerfSuite from NCSA
- From NCSA (Rick Kufrin) for use on the Itanium 2 TeraGrid IA64 installations.

psprocess Screenshot



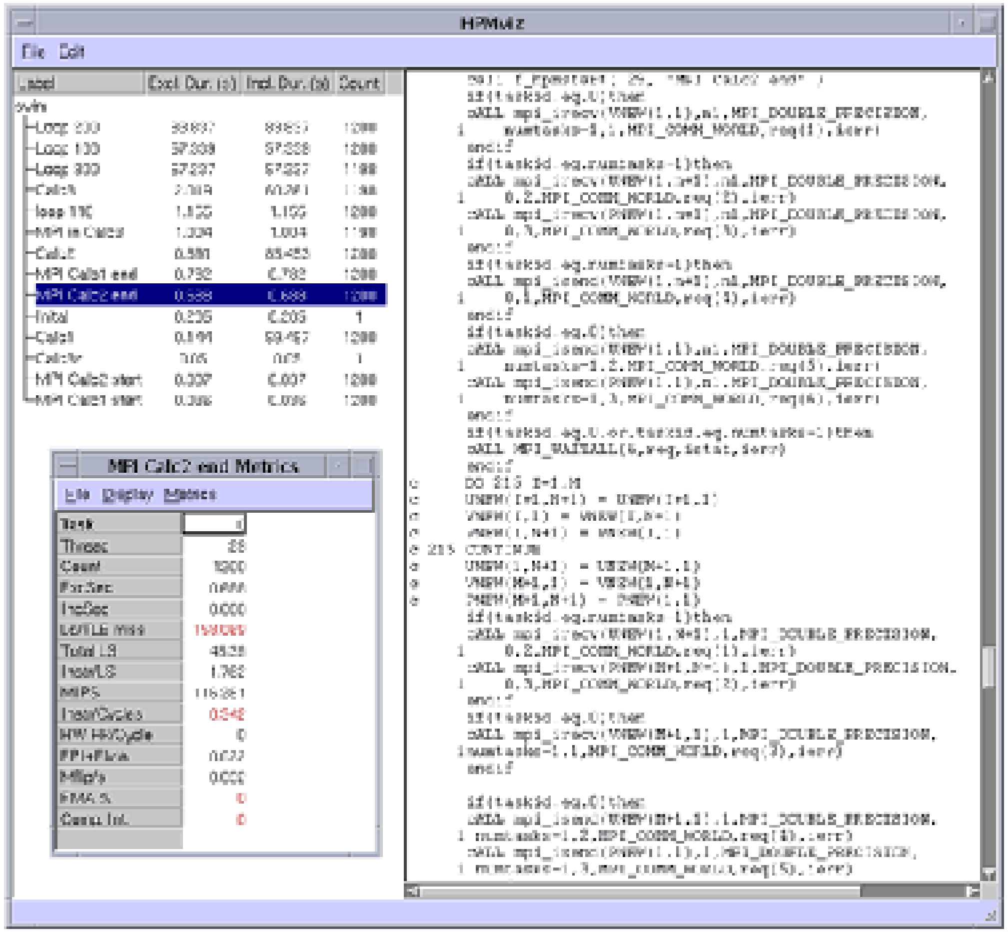
PerfSuite Hardware Performance Report

Derived Metrics	
[illegible]	[illegible]
Graduated floating point instructions per cycle	[illegible]
Floating point percentage of all graduated instructions	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
Data references per instruction	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
Percentage of cycles waiting for memory access	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

HPMToolkit

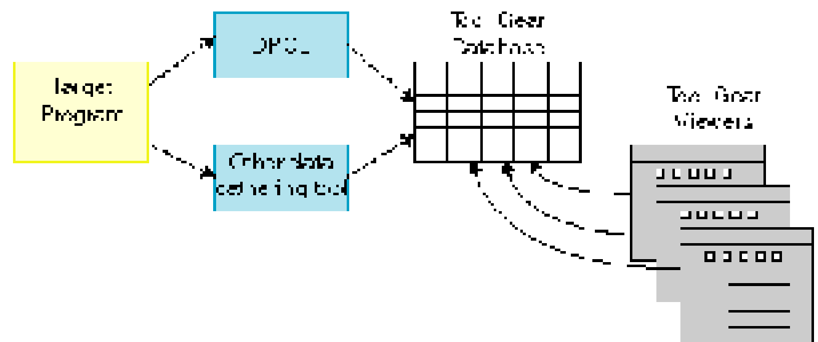
- Command line utility to gather aggregate counts.
  - PAPI version has been tested on IA32 & IA64
  - User can manually instrument code for more specific information
  - Reports derived metrics like SGI's perfex
- Developed by Luis Derose at IBM ACTC.
- Hpmviz is a GUI to view results

hpmviz Screenshot

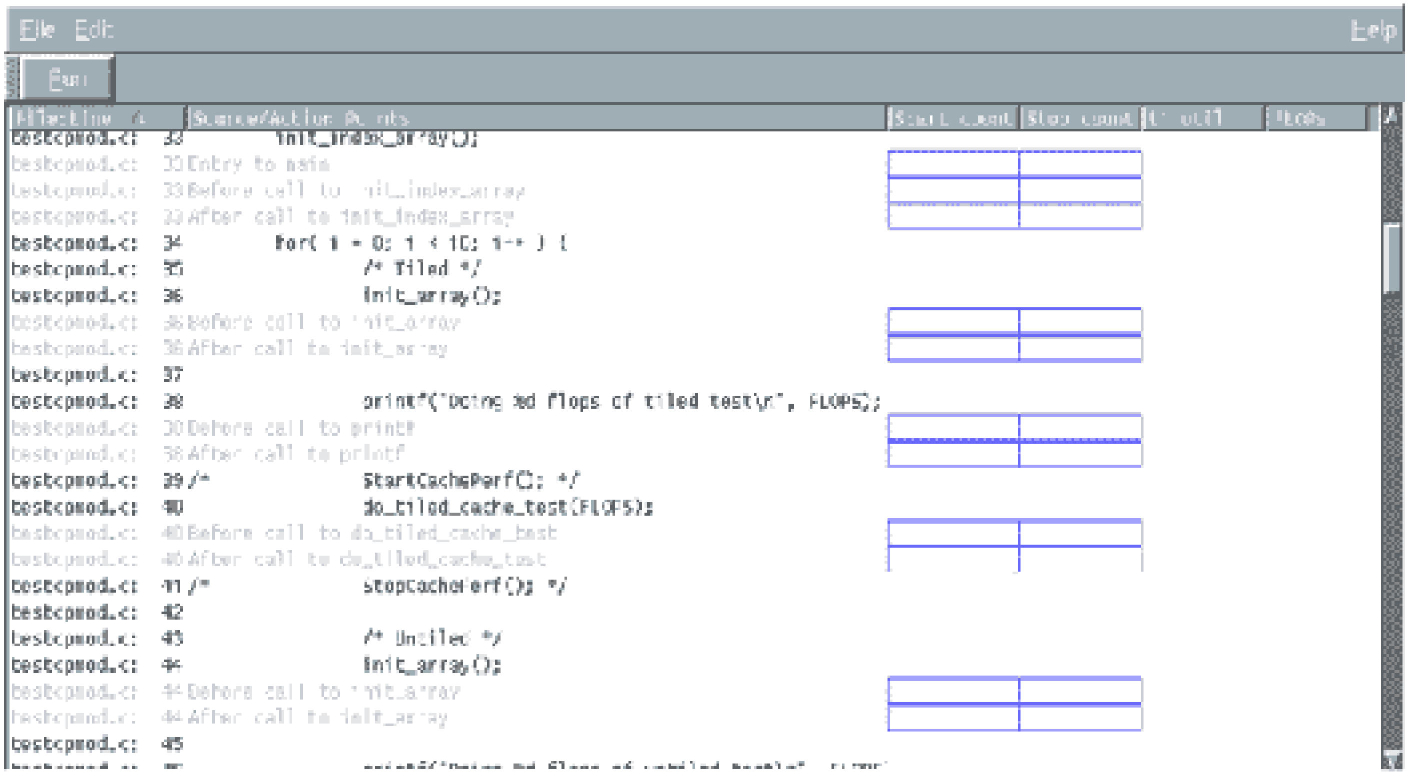


ToolGear

- Dynamic instrumentation and analysis suite from LLNL
- Based on DPCL
  - Works only on AIX/Linux
- Front end can accept data from ‘ any’ source



ToolGear Screenshot



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

DPCL vs. DynInst

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

- Available on all HPC platforms (and Windows)
- Breakpoints
- Arbitrary ins. points
- 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/

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

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
```

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: $T_{child} = \mathit{Inclusive}\ T_{function}$

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	Cal
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	Sub
TOTAL	100	1.723e+09	0
calc2	39.42	6.793e+08	168
calc1	35.28	6.08e+08	180
calc3	22.87	3.942e+08	165

1-Level Inclusive Call Tree of Metric PAPI_TOT_INS.

Parent/-Child	Percent	Total	Cal
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
- fsav	0.01848	1.255e+05	120
- mpi_irecv	0.02804	1.904e+05	120
- mpi_irecv	0.02804	1.904e+05	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

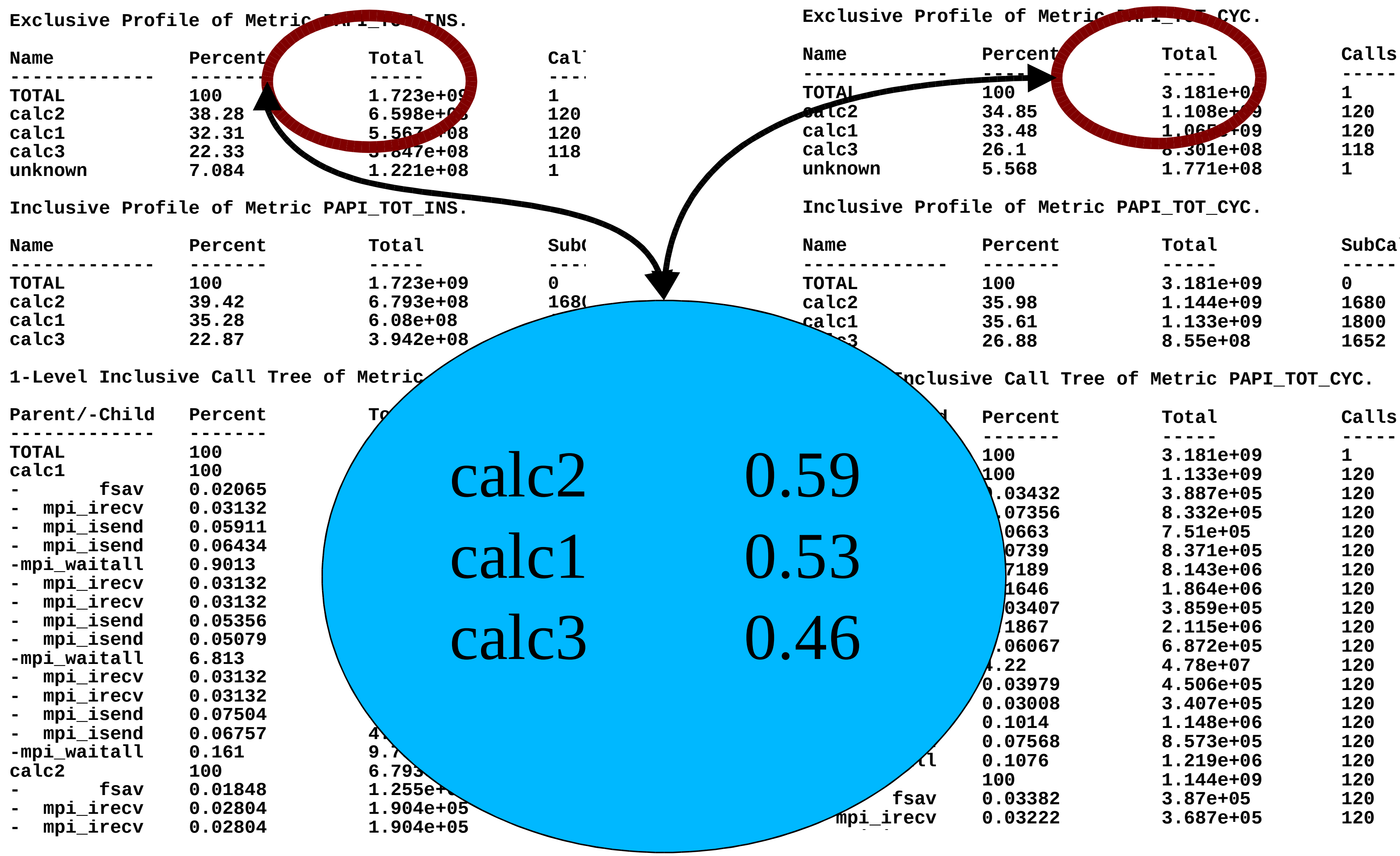
Inclusive Profile of Metric PAPI_TOT_CYC.

Name	Percent	Total	SubCa
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
- fsav	0.03382	3.87e+05	120
- mpi_irecv	0.03222	3.687e+05	120
- mpi_irecv	0.03554	4.067e+05	120

Swim Benchmark: Instructions per Cycle



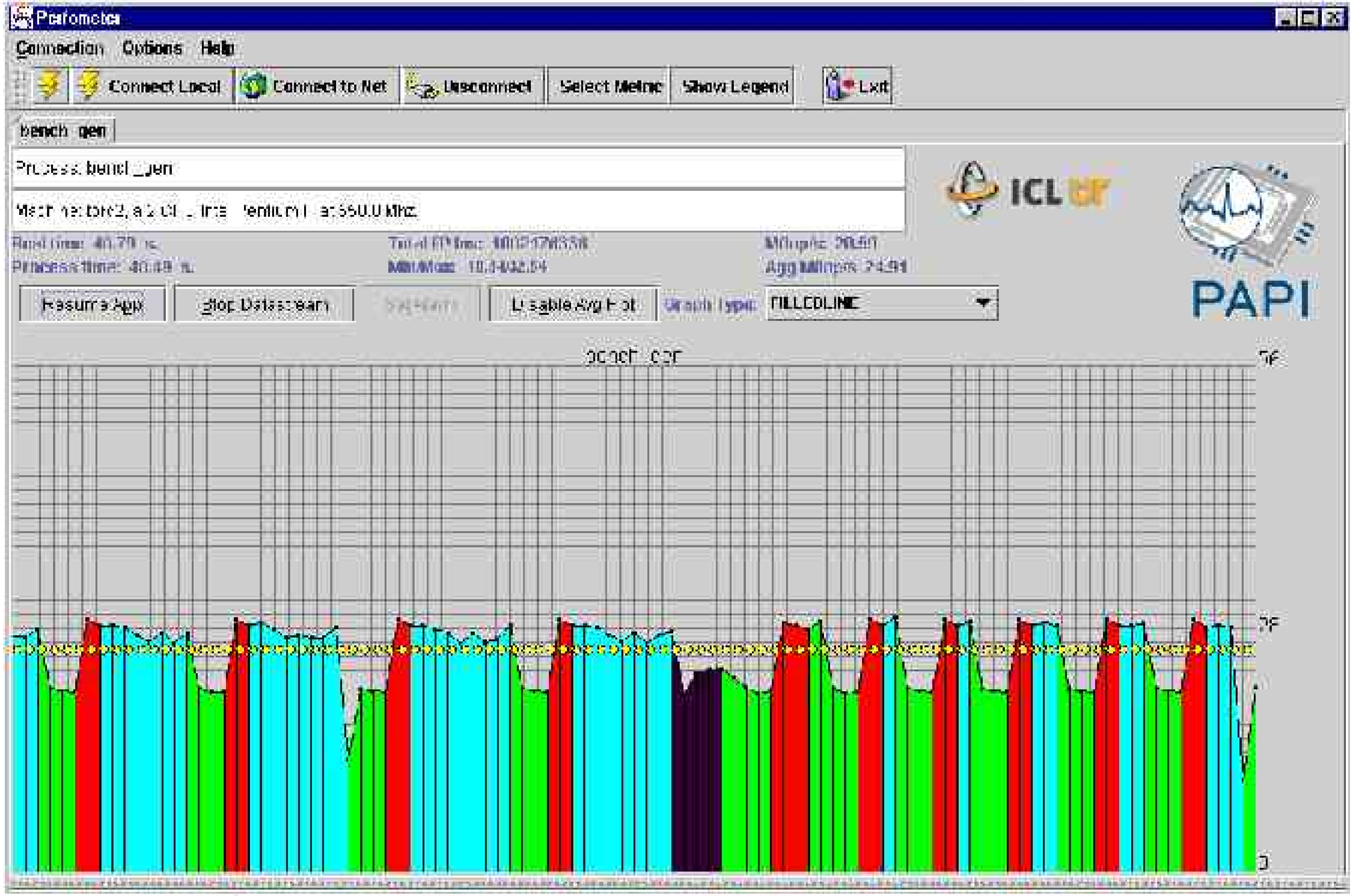
Perfometer Probe

- Graphically monitor rates of an applications performance in near real time.
- Ability to adjust metrics of interest as the application runs.
- Ability to mark functions of interest with different colors.

SWIM with perfometerprobe

```
Module perfometerprobe.so was loaded.
Module libperfometer.so was loaded.
Module libpapi.so was loaded.
(dynaprof) instr function swim.F calc1_ 0xff0000
swim.F, inserted 1 instrumentation points
(dynaprof) instr function swim.F calc2_ 0x00ff00
swim.F, inserted 1 instrumentation points
(dynaprof) instr function swim.F calc3_ 0x0000ff
swim.F, inserted 1 instrumentation points
(dynaprof) run
Module libnss_files.so.2 was loaded.
Module libnss_nisplus.so.2 was loaded.
Module libnsl.so.1 was loaded.
Module libnss_dns.so.2 was loaded.
Module libresolv.so.2 was loaded.
Perfometer client awaiting connection on port #33733
```

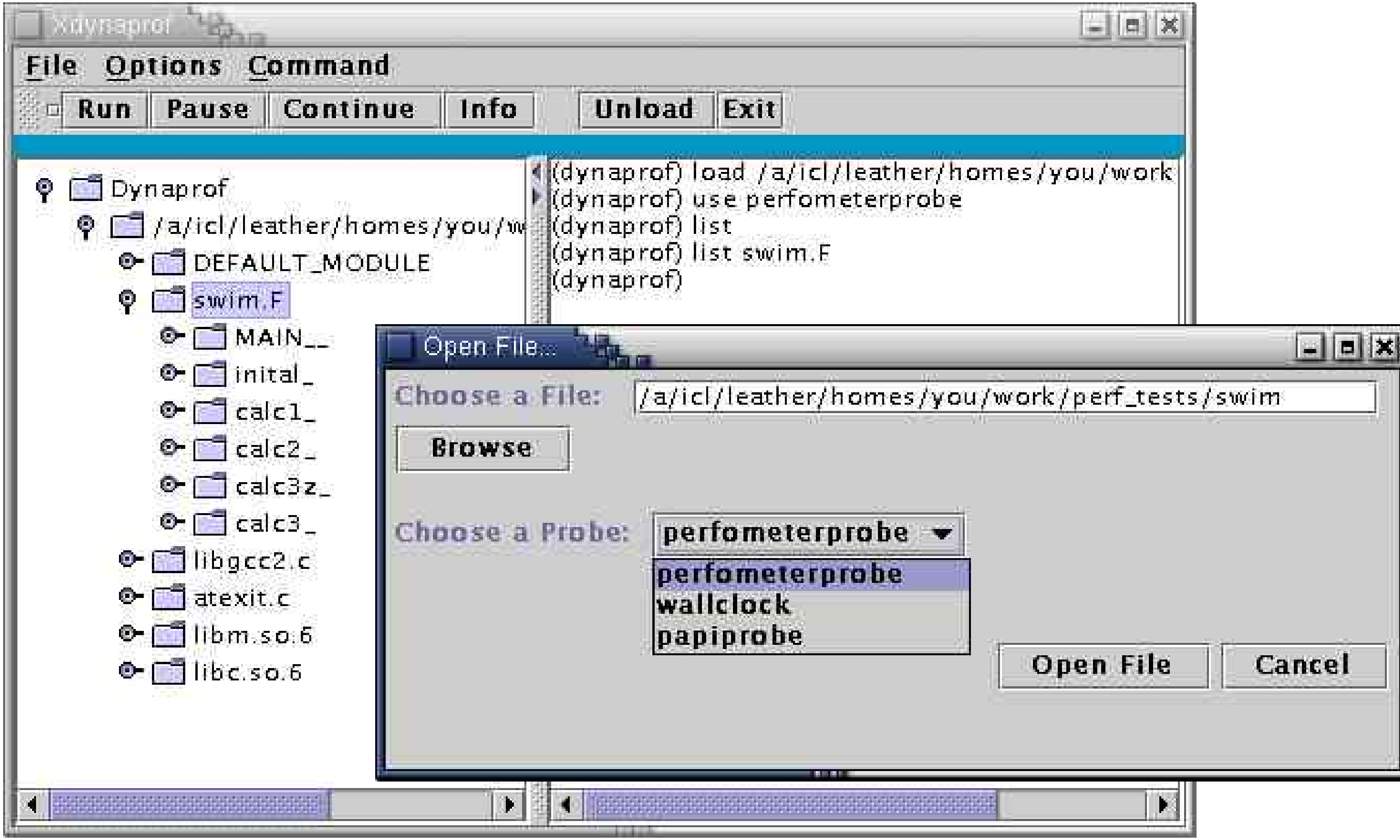
Perfometer Screenshot



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 GUI Screenshot



Dynaprof v0.8 Release

- Supported Platforms
  - Using DynInst
    - Linux 2.x
    - AIX 4.3/5?
    - Solaris 2.8
    - IRIX 6.x
  - 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

perfapi-devel@ptools.org

Dynaprof Future

- Changes to the DynInst version:
  - Port performance probes to IA64
  - Interactive instrumentation of MPI codes, faster start-up
  - Integration with the Tool Daemon Protocol from U. Wisconsin.
  - New instrumentation point support
    - CFG/Basic Block/Loop level instrumentation exists but is untested
    - Arbitrary start/stop points
  - Arbitrary breakpoints and 'run-until' support
  - Support for programs that dynamically load extensions (i.e. Mozilla)
- Documentation:
  - Probe API
  - Tutorial
- Probes
  - **Integration with TAU:**
    - **tauprobe**
    - **TAU/jracy compatible output from papiprobe and wallclockprobe**
  - Additional thread model support: sproc() on IRIX
  - Improved data structures for handling for multiple instrumentation cycles
  - Thread support in perfometer probe

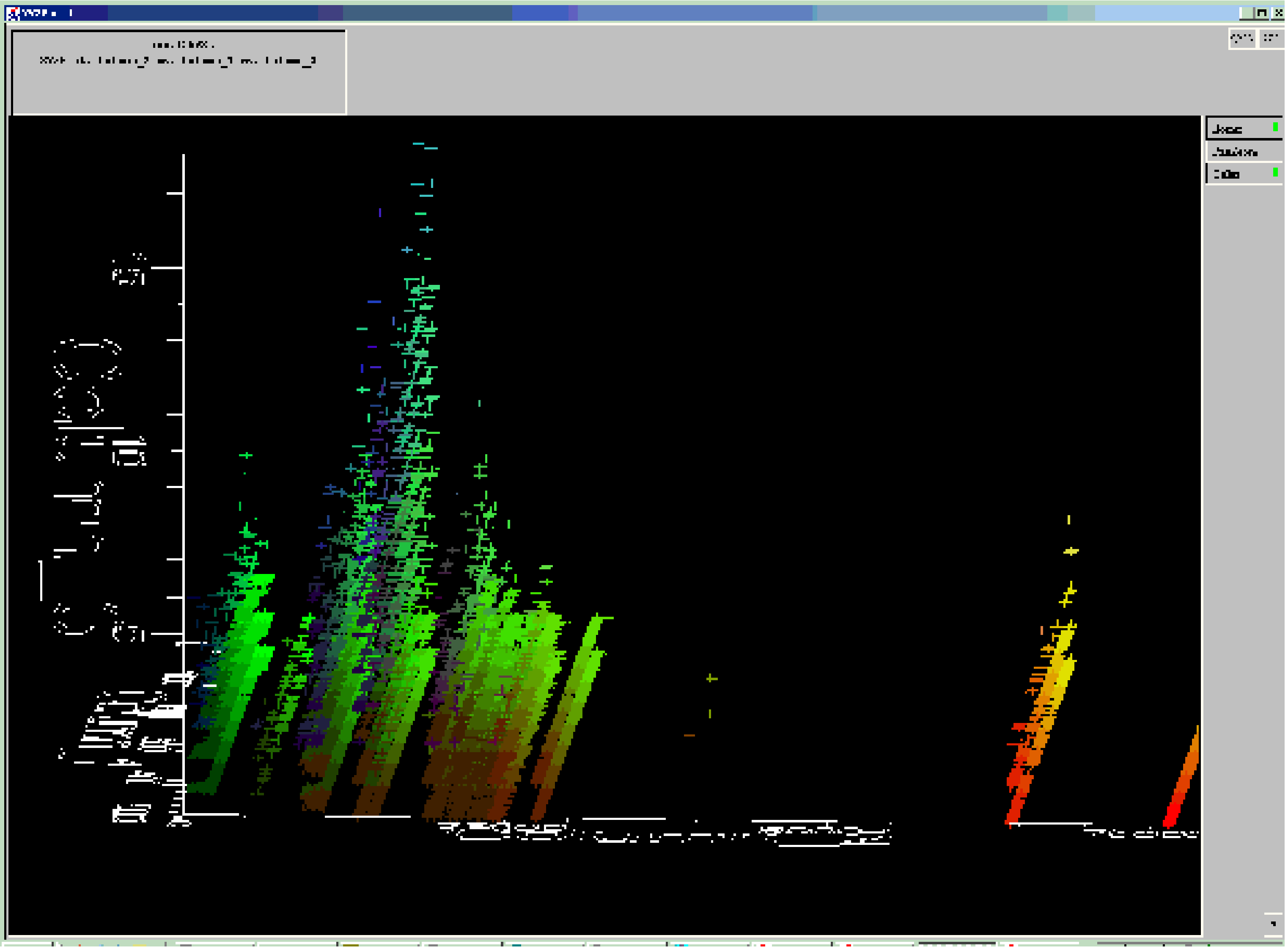
Phase Profiling

- Statistical profiling is often static
  - Gprof, Quantify, Speedshop, Workshop, Tprof, etc…
- We want to understand all aspects of a program's performance. What about behaviour over time?
- Applications vs. kernels have distinct phases.
  - Initialization
  - Data input
    - Compute
    - Communicate
    - Repeat
  - Data output
  - Finalization

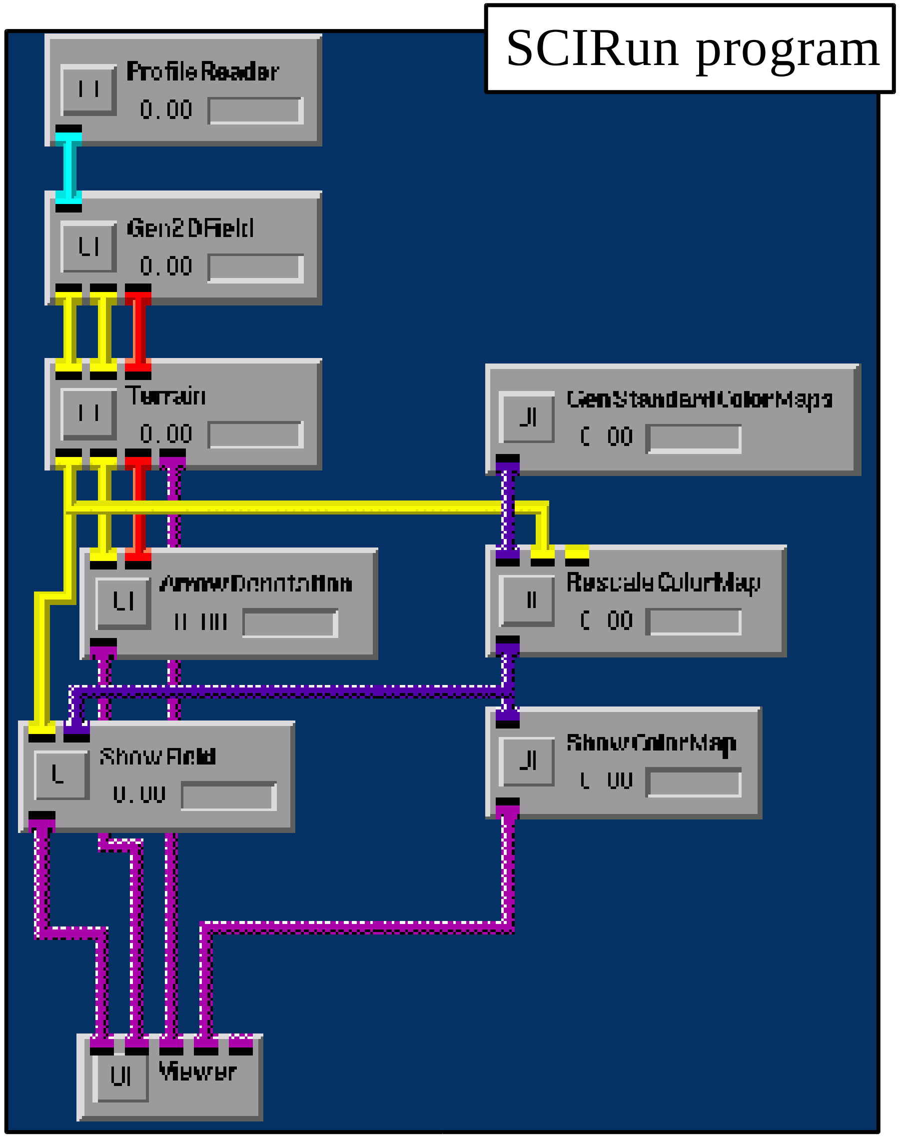
Phase Profiling 2

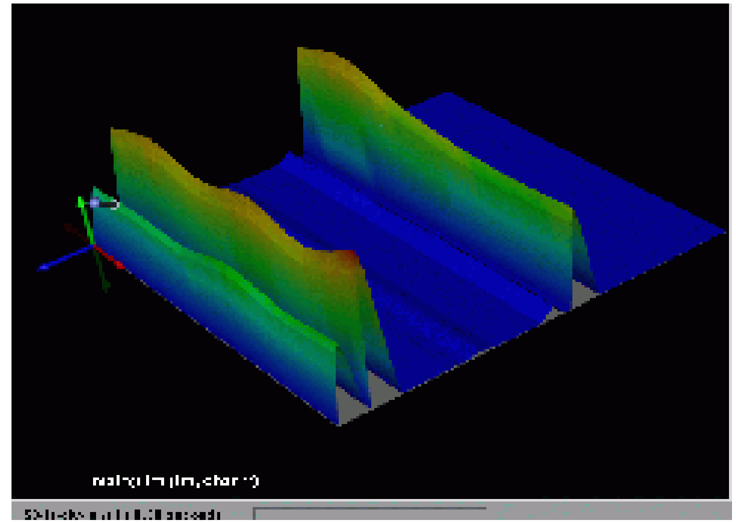
- Workloads on the hardware are most often periodic.
- Open questions:
  – How do we process, visualize and understand this data in a scalable fashion?
  – Can we use this data to optimize an application in the temporal domain?
  – Can we parameterize this data against *(t)* for performance models?

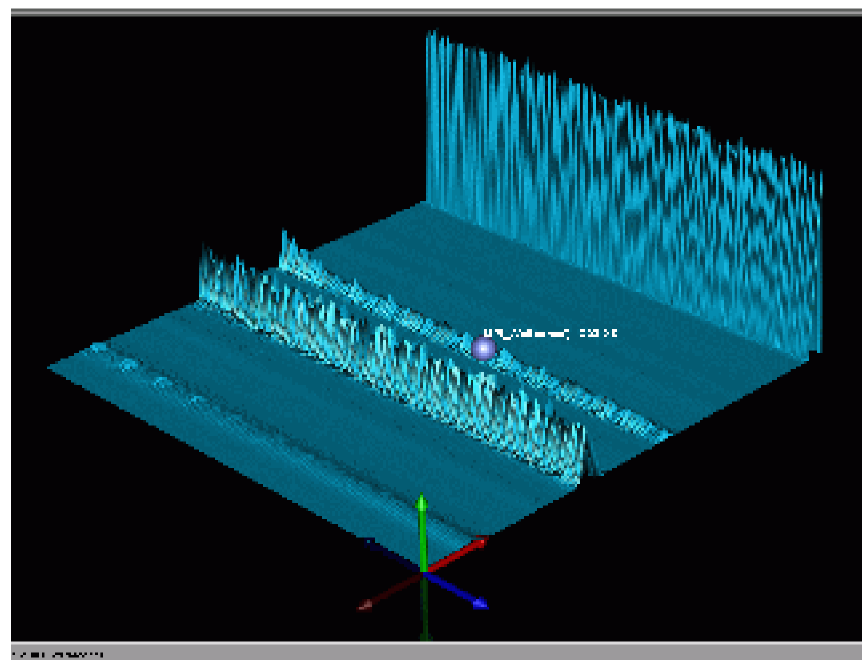
Sample of Space vs. Time vs. Frequency



2D Field Performance Visualization in TAU & SCIRun







Performance Tool Trends

- Most of the infrastructure now exists.
- Many sites are "rolling their own".
- Can one size fit all?
- 2 types of tools evolving:
  - Simple
  - Comprehensive

Performance Database

- Database of all relevant information regarding the performance of a code.
  - Source code structure
  - Transformations performed during optimization
  - Static and dynamic memory allocation information
  - Derived data types, etc…
- Examples:
  - TAU PDT: Program Database Toolkit
  - HPC Tools: XML Database
  - ToolGear
- This data can be quite large! Remember MPI traces?

Some problems to be solved

- How do we get the data out of the threads/processors/nodes/application?
  - Aggregation
  - Filtering
  - Reduction
  - Tool Daemon Protocol from U. Wisconsin
- How do we correlate performance data from optimized code to the exact line of source code?
  - Software pipelining
  - C++ Templates

WWW References

- http://icl.cs.utk.edu/projects/papi
- http://www.cs.utk.edu/~mucci/dynaprof
- http://www.cs.rice.edu/~dsystem/hpcview
- http://aros.ca.sandia.gov/~cljanss/perf/vprof
- http://www.alphaworks.ibm.com/tech/hpmtoolkit
- http://perfsuite.ncsa.uiuc.edu/psrun
- http://software.sci.utah.edu/scirun.html
- http://www.paradyn.org
- http://www.cs.uoregon.edu/research/paracomp/tau
- http://vibes.cs.uiuc.edu/Software/SvPablo/svPablo.htm
- http://www.aei-potsdam.mpg.de
- http://aros.ca.sandia.gov/~cljanss/perf/vprof
- http://www.llnl.gov/asci/projects/asde/toolgear.html
- http://www.dyninst.org
- http://oss.software.ibm.com/developerworks/opensource/dpcl