

# PAPI and Hardware Performance Analysis Tools

Center for Parallel Computers, Royal Institute of Technology  
Performance Workshop, May 26th, 2003

**Philip Mucci, Research Consultant**  
Innovative Computing Laboratory/UTK  
Performance Evaluation Research Center/LBNL  
[mucci@cs.utk.edu](mailto:mucci@cs.utk.edu)





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.

- PAPI
  - Quick Overview
  - 3.0 Feature Outline
- Performance Analysis Tools
- Trends

- 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...

- Small number of registers dedicated for performance monitoring functions.
  1. AMD Athlon, 4 counters
  2. Pentium  $\leq$  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 Diagram

Figure 1: POWER4 Chip Logical View

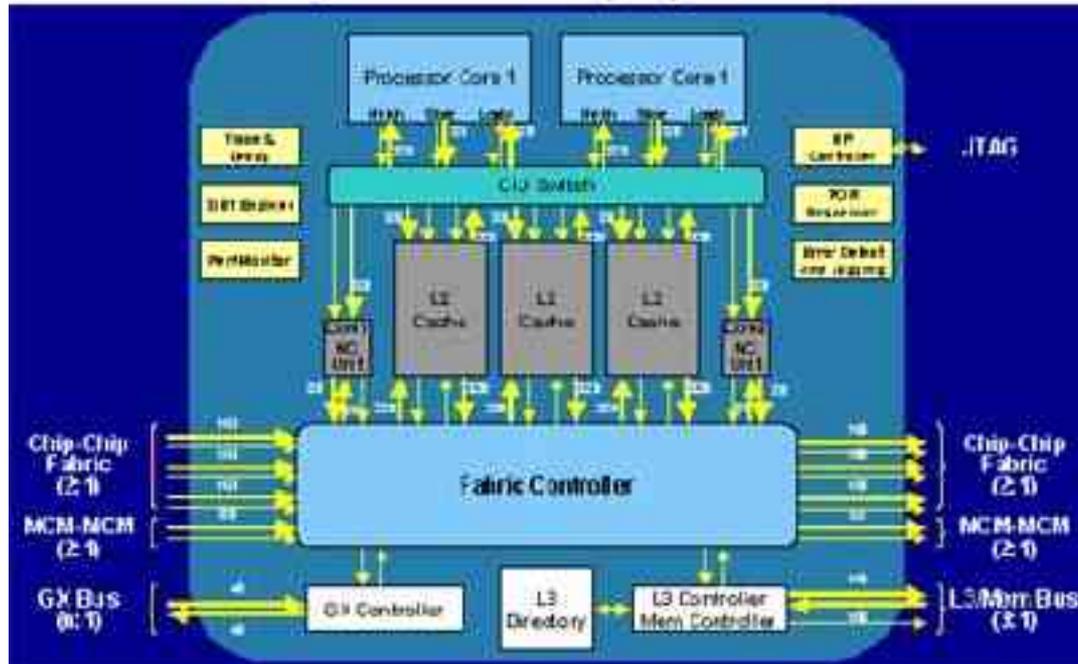
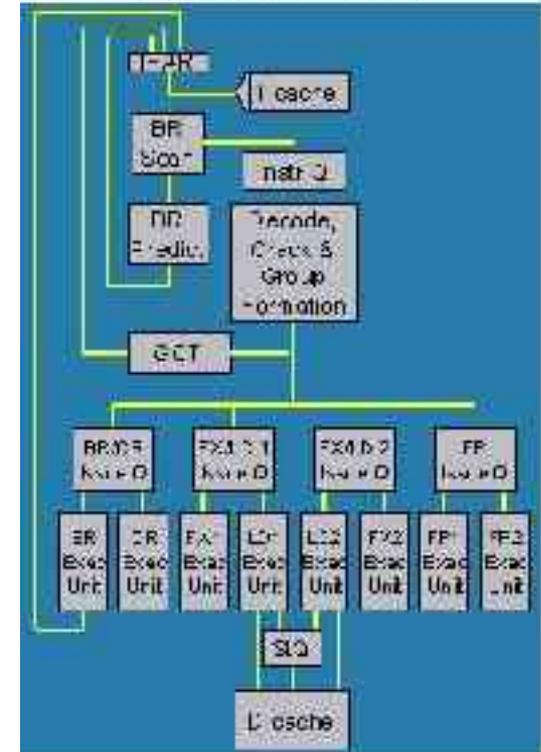
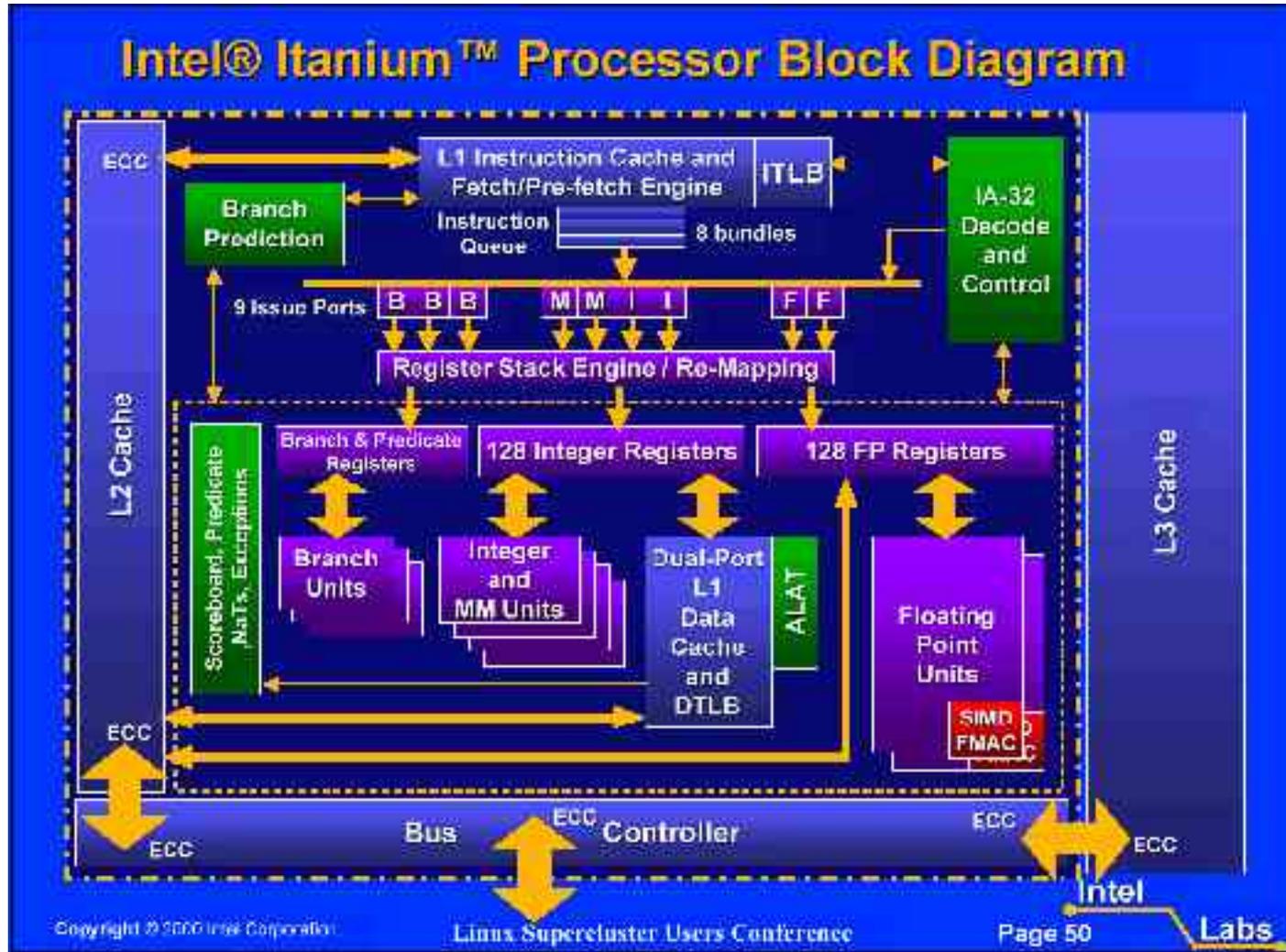


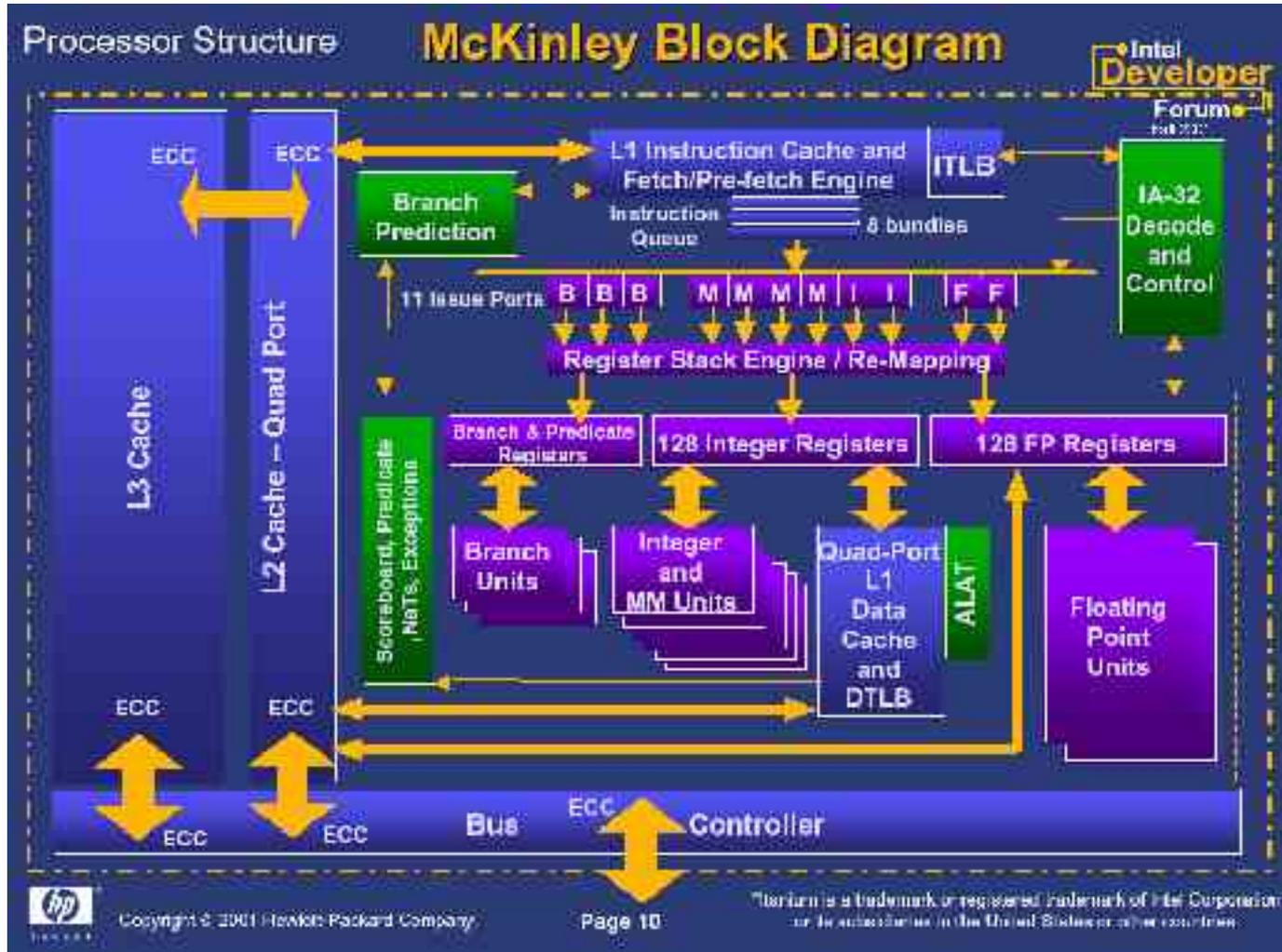
Figure 2: POWER4 Core



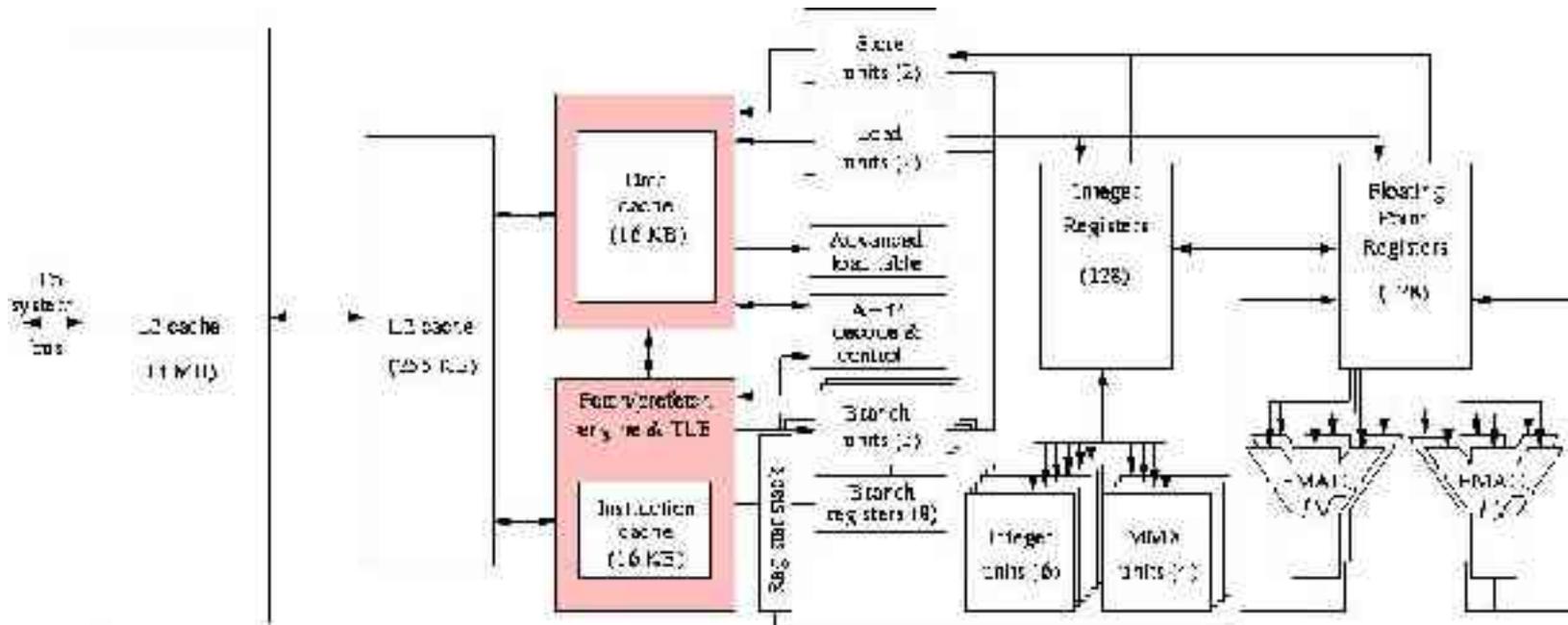
# Itanium 1 Block Diagram



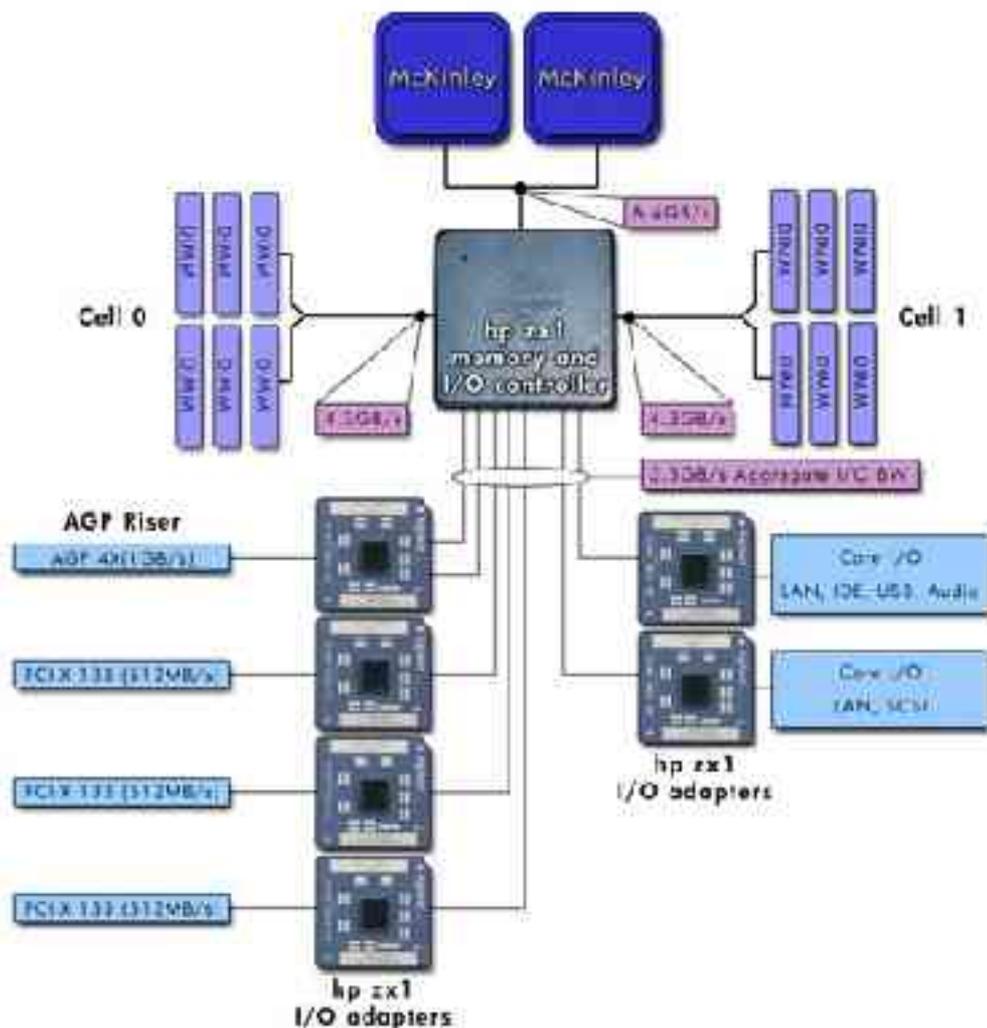
# Itanium 2 Block Diagram



# Itanium 2 Block Diagram

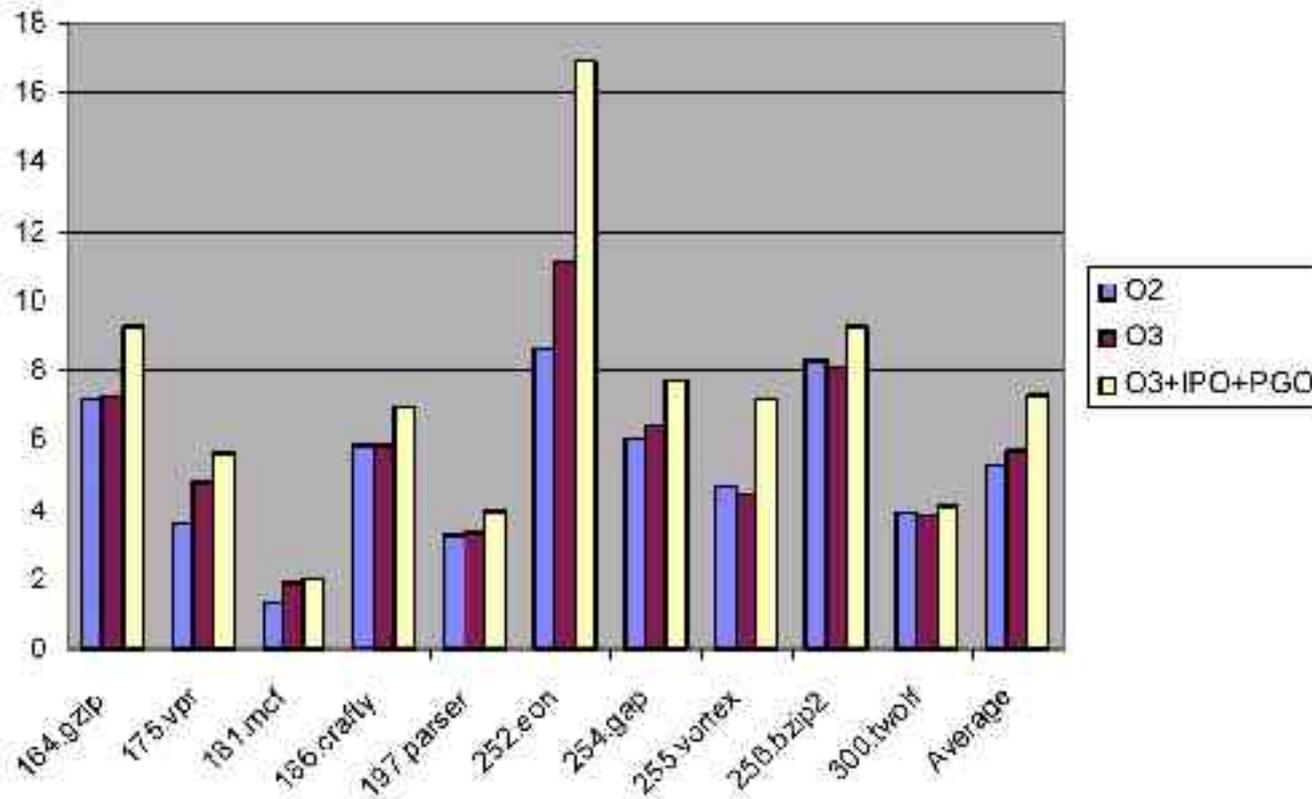


# PDC Itanium 2 System Architecture

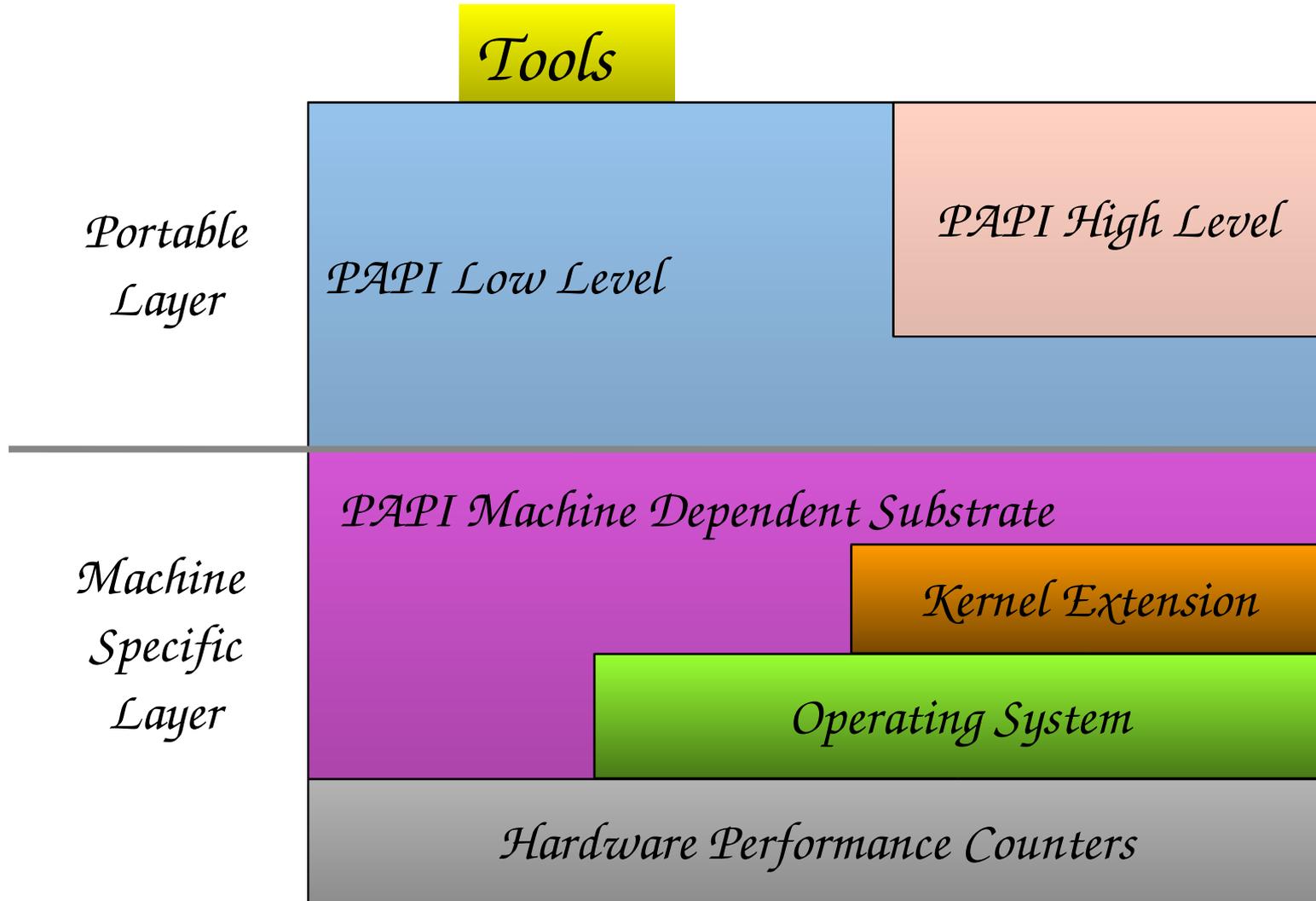


# Importance of Optimization

## Example: Speed up from Static Compiler Optimization on Itanium-1 in 2002 (SpecInt)



# PAPI Implementation



- 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

- 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**

> 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 cache line
PAPI_CA_CLN	0x8000000b	No	No	Requests for clean cache line
PAPI_CA_INV	0x8000000c	No	No	Requests for cache line inv.



- 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

- 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

- 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](http://icl.cs.utk.edu/projects/papi/files/html_man/papi.html#4))
- Thread-safe (SMP, OpenMP, Pthreads)
- Supports both presets and native events

- 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

- 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.

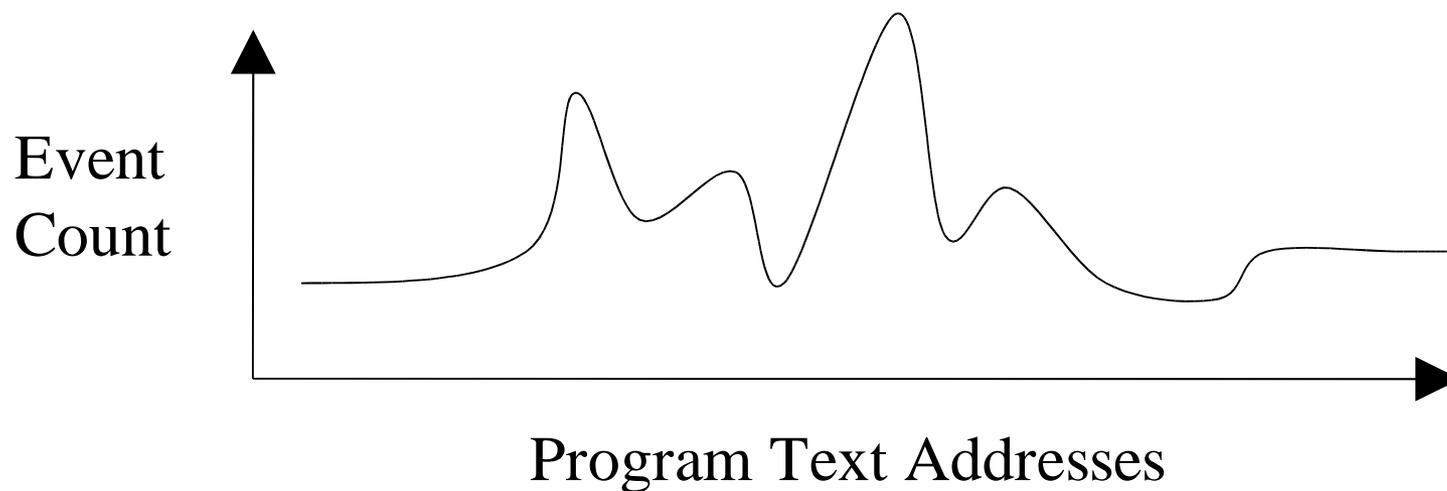
# Interrupts on Counter Overflow

---

- 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.

- 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.

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

- 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, 4
- Enhancements
  - Static/dynamic  
memory info
  - Multiplexing  
improvements
  - Lots of bug fixes

- Using lessons learned from years earlier
  - Substrate code: 90% used only 10% of the time
- I Want to formalize the API during this visit!
- Redesign for:
  - 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, while register access alone costs 100 cycles.
- Full native and programmable event support
  - Thresholding
  - Instruction matching
  - Per event counting domains

- 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

- 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 API
  - Process footprint

- System-wide and process wide counting implementation
- High level API made thread safe
- New language bindings
  - Java
  - Lisp
  - Matlab

- First release expected Summer, 2003
- Additional platforms
  - Cray X-1
  - AMD Opteron/K8
  - Nec SX-6
  - Blue Gene (BG/L)



# PAPI Tools

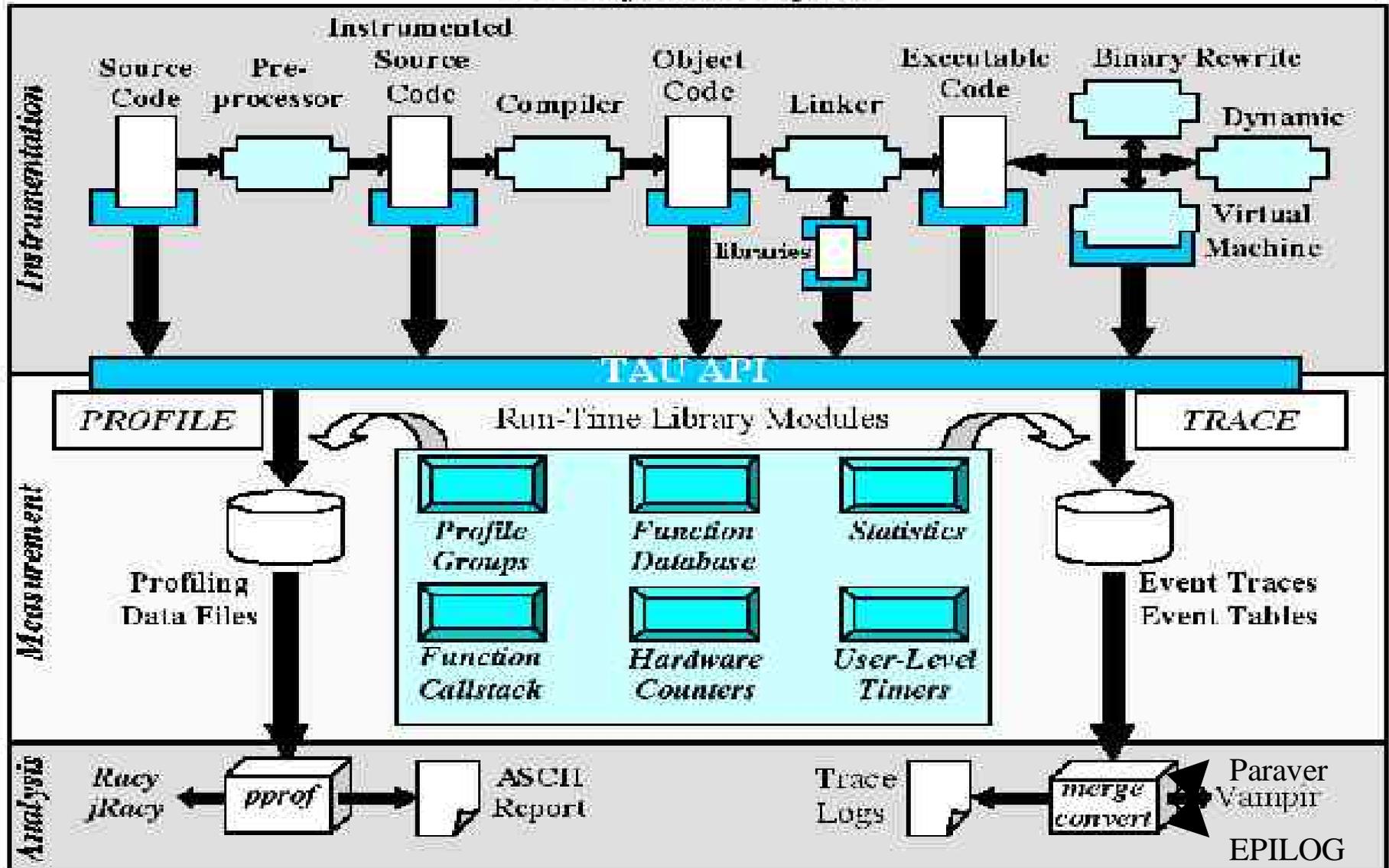
---

- 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
  - Cache misses

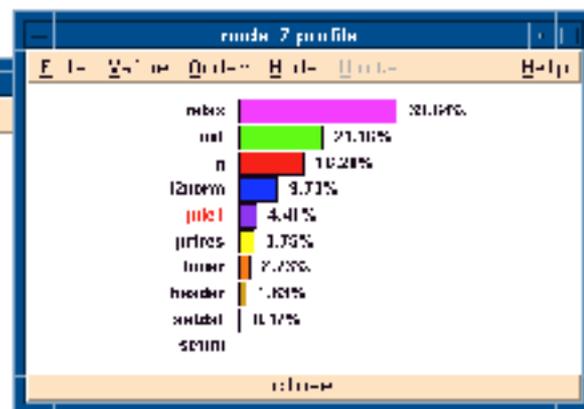
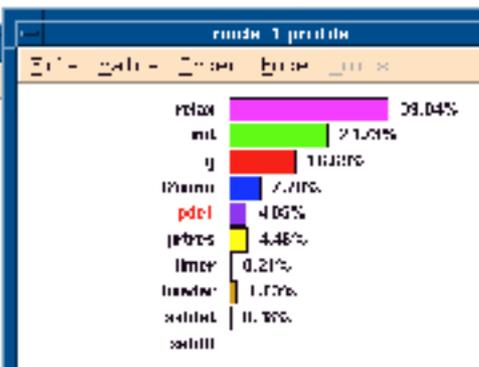
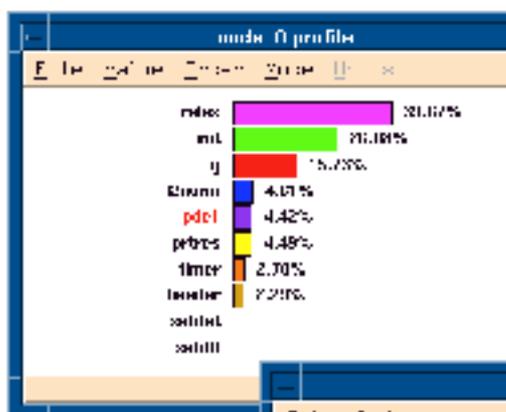
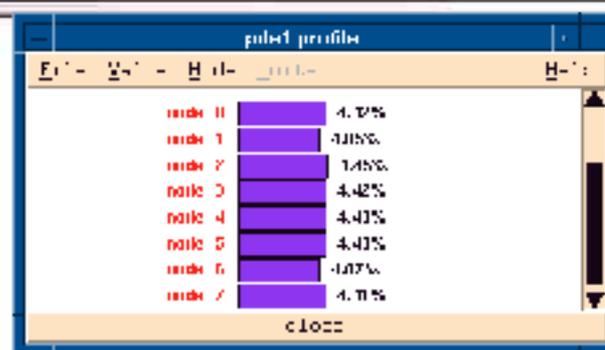
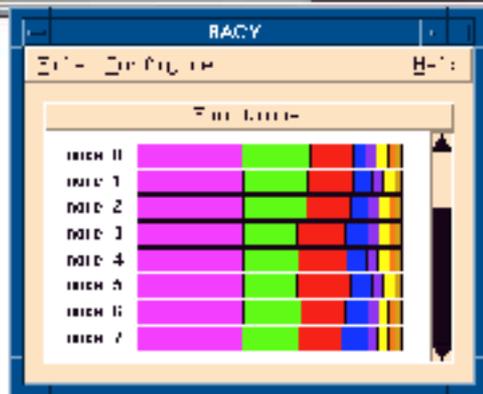
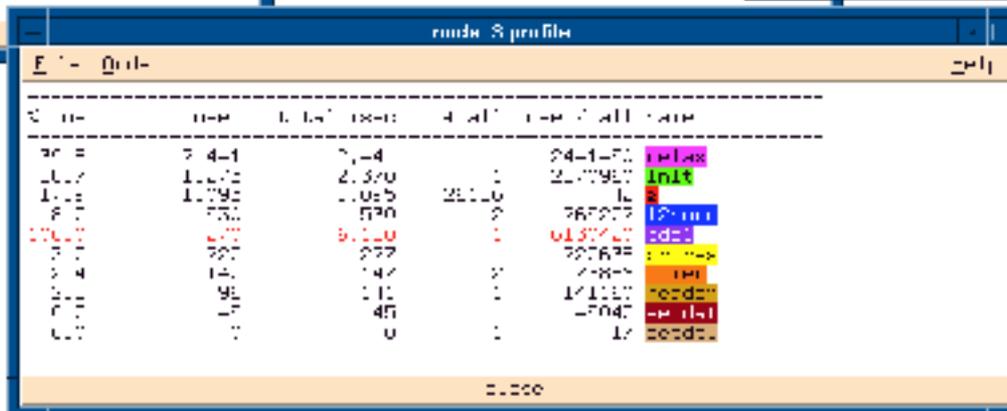
- From Allen Maloney's Group at U. Oregon
- Source or binary based
- Different visualization plugins
- Supports all forms of parallelism
- Integration with Vampir

# TAU Performance System Architecture

TAU Performance System



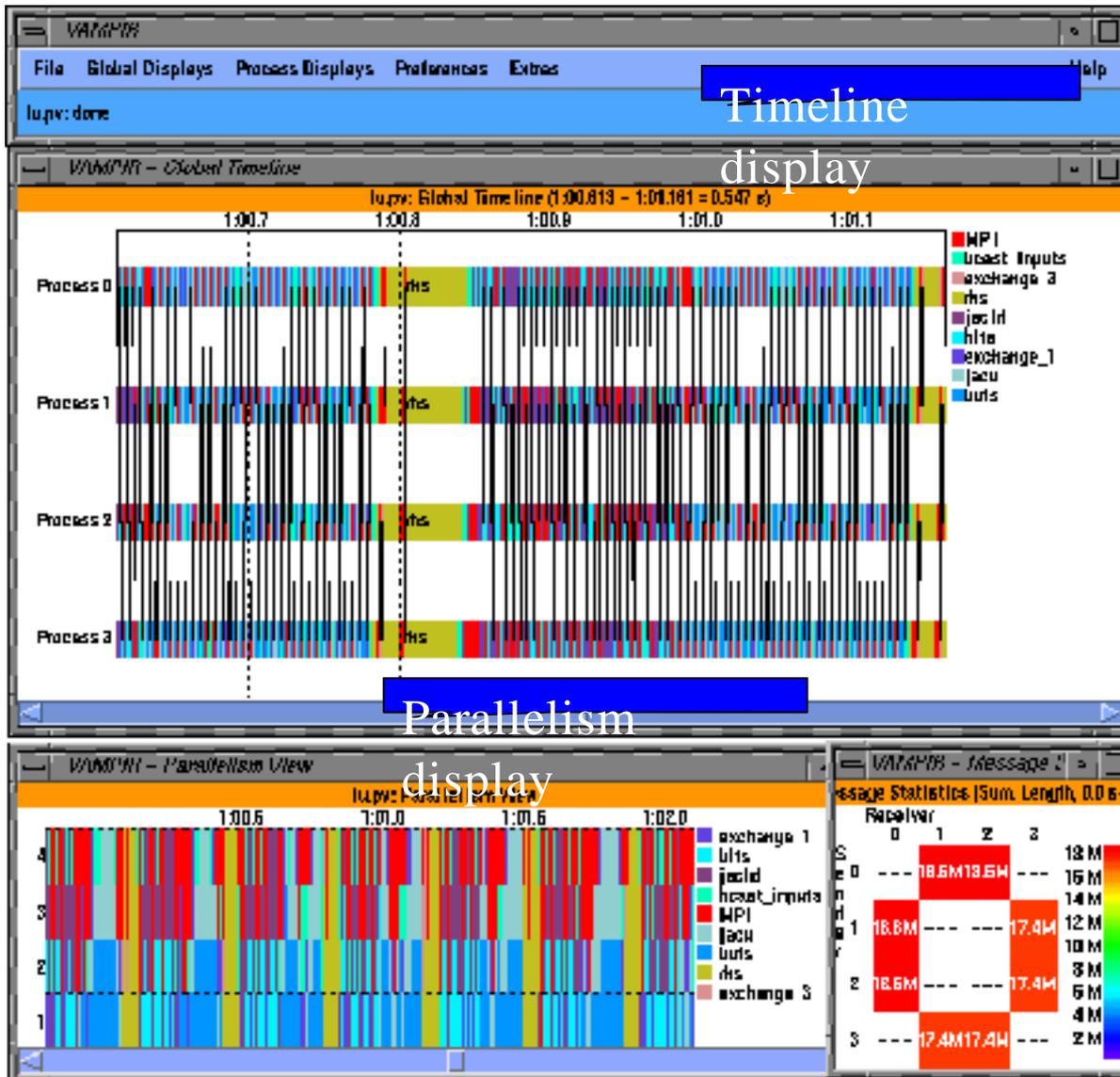
# TAU Screenshot

run 8 profile window showing a table of function call statistics:

Call	run	total calls	total time	total time / call time	Function
relax	24-1	24	24.000	1.000	relax
init	24-1	24	24.000	1.000	init
g	24-1	24	24.000	1.000	g
l2norm	24-1	24	24.000	1.000	l2norm
pk1	24-1	24	24.000	1.000	pk1
prfres	24-1	24	24.000	1.000	prfres
timer	24-1	24	24.000	1.000	timer
header	24-1	24	24.000	1.000	header
selknt	24-1	24	24.000	1.000	selknt
selknt	24-1	24	24.000	1.000	selknt

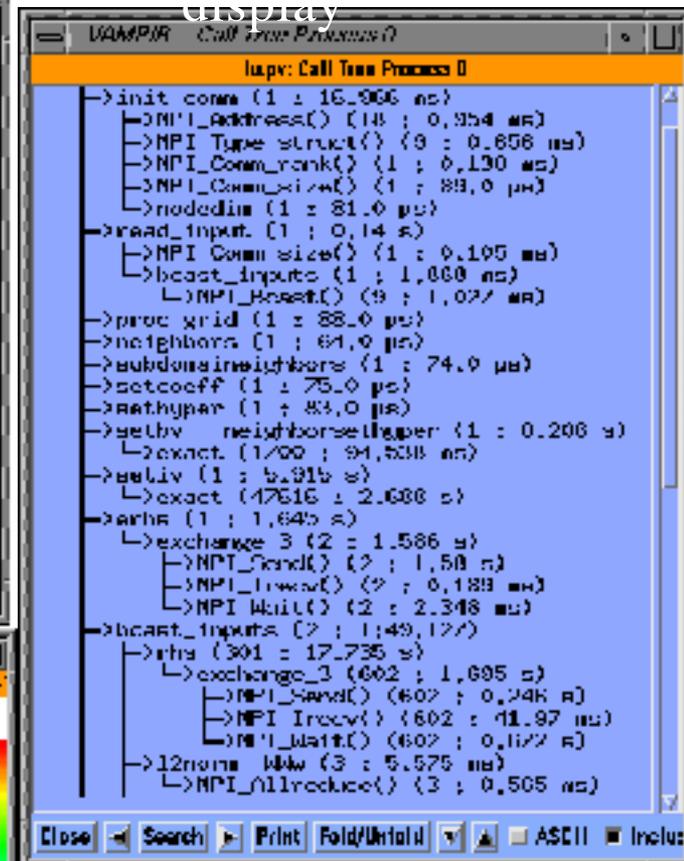
# Vampir (NAS Parallel Benchmark – LU)



Timeline display

Parallelism display

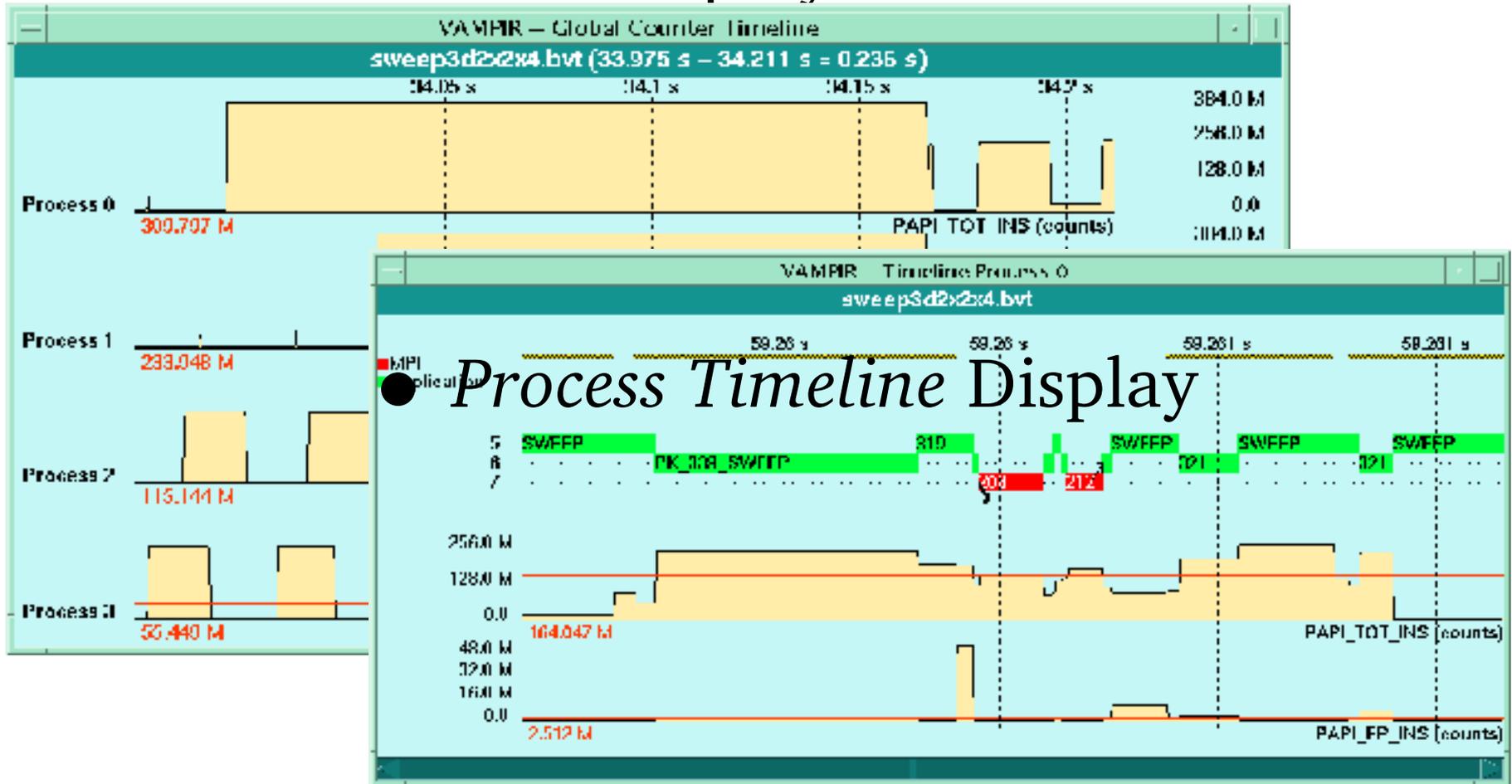
Callgraph display



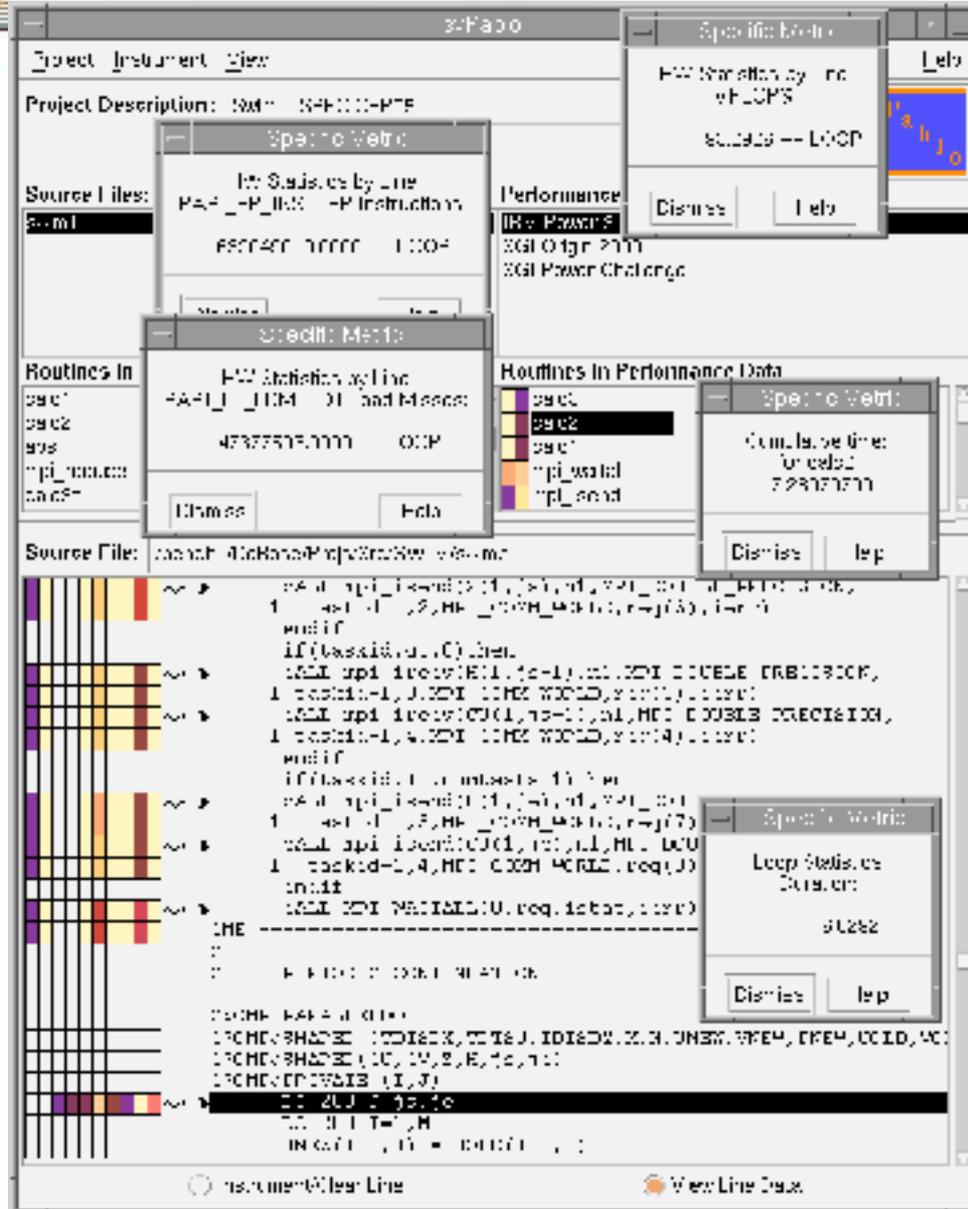
Communications

# Vampir v3.x: HPM Counter

- *Counter Timeline Display*



# SvPablo from UIUC



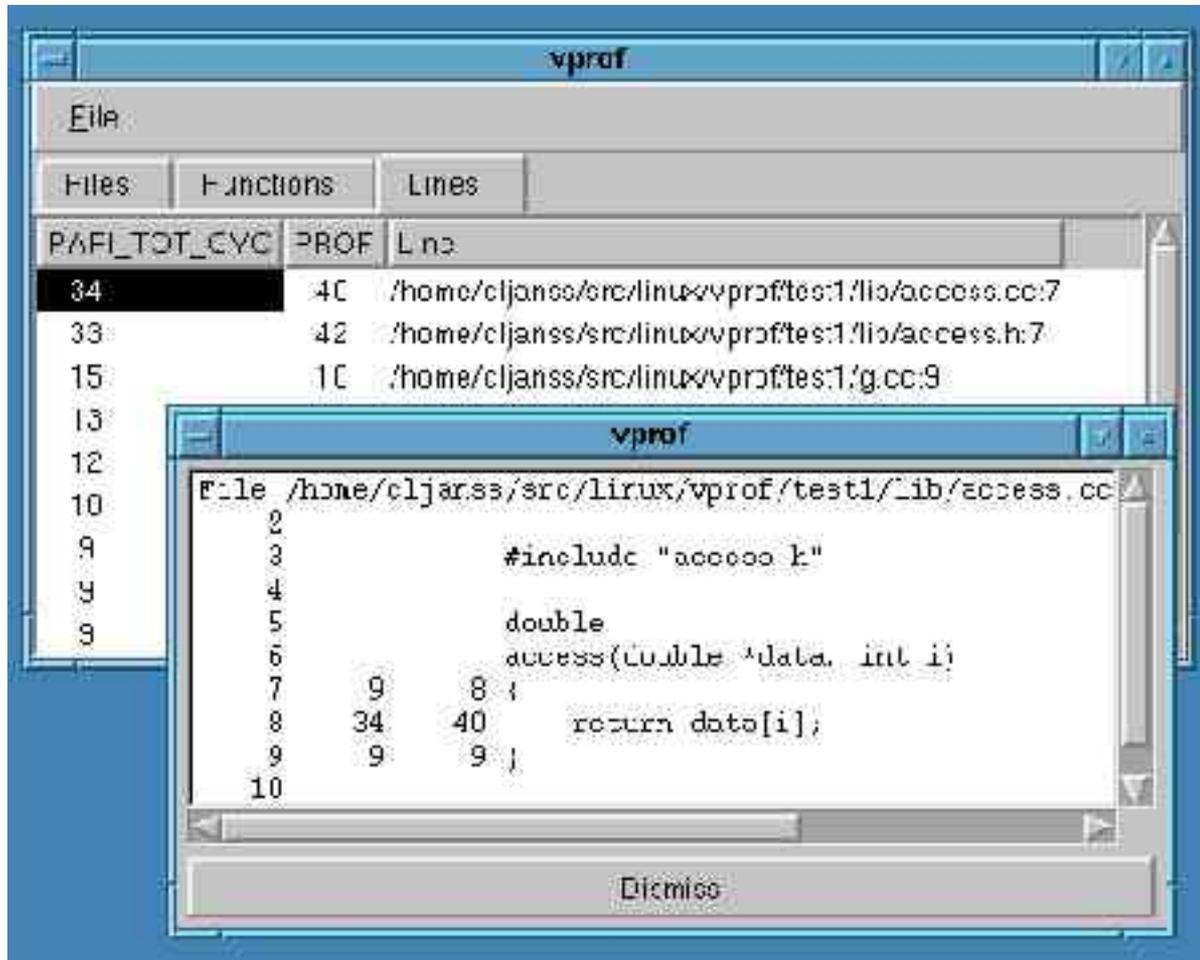
The screenshot displays the SvPablo application window. At the top, it shows 'Project Description: IBM - SPARC-PRE' and 'Source Files: s-m1'. The main window is divided into several panes:

- Performance:** Displays 'IBM POWER9', 'EGL O3pr 200', and 'EGL Power Challenge'.
- Routines in Performance Data:** A list of routines including 'spc1', 'spc2', 'spc3', 'mpi\_wait', and 'mpi\_send'.
- Source File:** Shows the file path 'ibmarch-40Knode-Proj-SPARC-PRE-s-m1.f90'.
- Main Window:** Displays Fortran source code with a color-coded performance overlay on the left side, indicating execution time for different lines.
- Dialog Boxes:** Several 'Specific Metric' dialog boxes are open, showing various performance metrics such as 'Loop Statistics by Line', 'Loop Statistics by Line', and 'Loop Statistics by Line'.

At the bottom of the window, there are radio buttons for 'Instrument Clear Line' and 'View Line Data'.

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

# Vprof from Sandia National Laboratory



The screenshot shows the vprof application interface. The main window displays a list of files and functions with columns for PAFI\_TOT\_CYC, PROF, and Line. A smaller window is open over the main one, showing the source code for the function 'access' in 'access.c'.

PAFI_TOT_CYC	PROF	Line
34	40	/home/cljanss/src/linux/vprof/test1/lib/access.c:7
33	42	/home/cljanss/src/linux/vprof/test1/lib/access.h:7
15	10	/home/cljanss/src/linux/vprof/test1/g.c:9

```

File /home/cljanss/src/linux/vprof/test1/lib/access.c
2
3      #include "access.h"
4
5      double
6      access(double *data, int i)
7      {
8          return data[i];
9      }
10
  
```

- 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

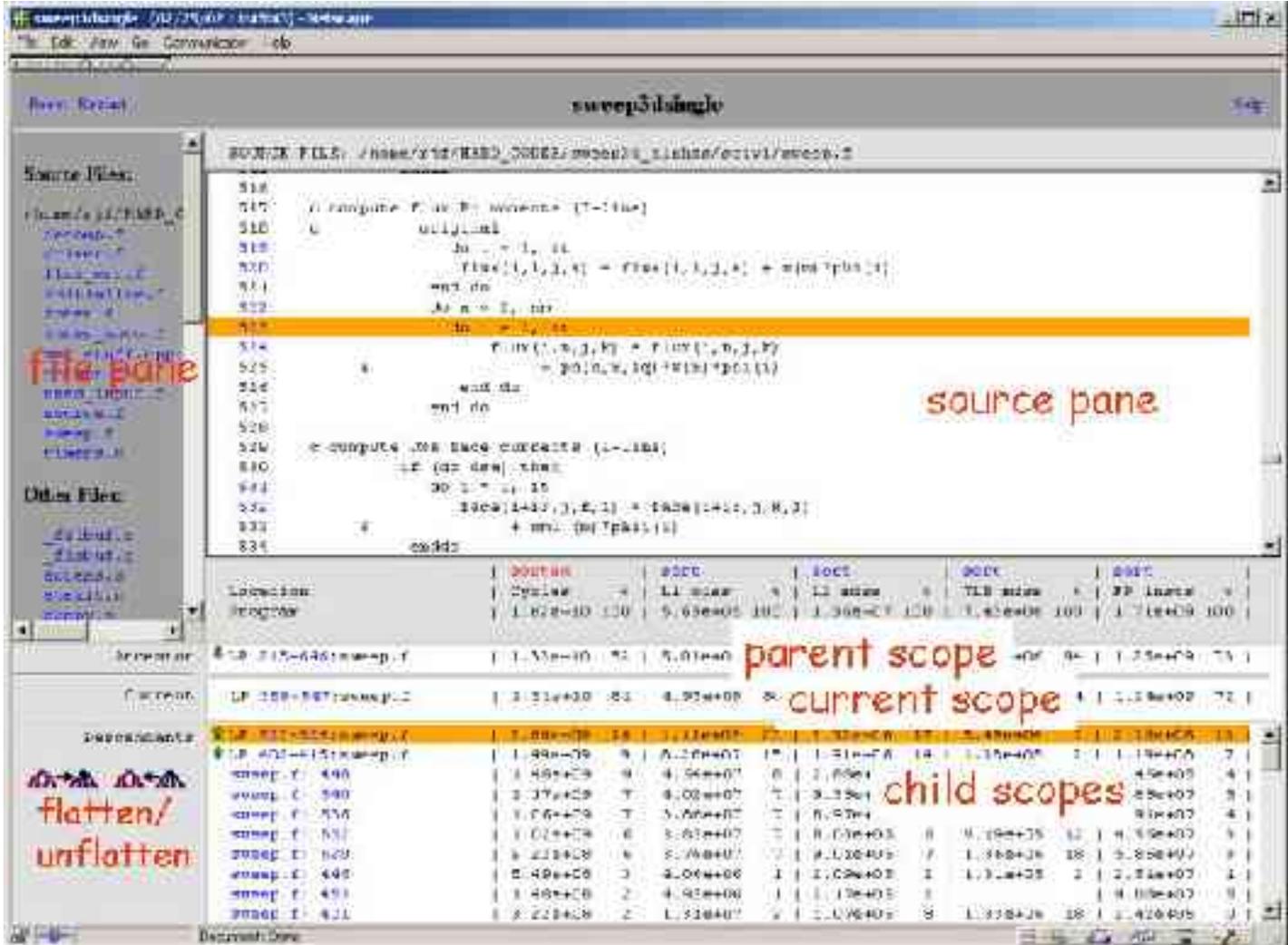
<http://aros.ca.sandia.gov/~cljanss/perf/vprof>



- 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>

- 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



The screenshot displays the HPCView interface with the following components:

- Source Files:** A tree view on the left showing the file structure, including a **file pane** label.
- Source Code:** A central pane showing Fortran code for a program named `sweep3dsingle`. A **source pane** label is present.
- Scope Hierarchy:** A table below the code showing the execution context. It includes a **parent scope** and **current scope** label. The table lists various scopes with their locations and program names.
- Performance Metrics:** A table at the bottom showing execution statistics for different scopes, including time and memory usage. A **child scopes** label is present.
- Control Elements:** Icons for **flatten/unflatten** are visible in the bottom left corner.

Location	Program	Start	End	Start	End	Start	End
0x00000000							

Scope	Location	Program	Start	End	Start	End	Start	End
Parent Scope	0x00000000							
Current Scope	0x00000000							
Child Scope 1	0x00000000							
Child Scope 2	0x00000000							

- 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.

- 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>

- `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

## PerfSuite Hardware Performance Report

Hardware Metrics	
Guests per instruction per cycle	1.785
Guests per floating point instruction per cycle	0.115
Floating point percentage of all produced instructions	3.21%
Guests per bus bandwidth cycle	0.715
Guests per bus bandwidth floating point instruction	1.14
Execute time per instruction	0.025
Ratio of floating point instructions to floating point cycles	1.94
L1 instruction cache miss ratio	0.008
L1 data cache miss ratio	0.002
L1 hit to cache miss ratio	11.25
L2 cache read hit ratio	0.992
L2 cache read hit with ratio	0.991
Ratio of instructions to memory predicted branches	0.030
L1 cache read hit ratio	0.997
L2 cache read hit ratio	0.457
L3 cache read hit ratio	0.71
L1 cache hit ratio (instr)	0.48
L2 cache hit ratio (instr)	0.997
L3 cache hit ratio (instr)	0.991
Branches used to L2 cache (MIPS)	1002.391
Branches used to L3 cache (MIPS)	1000.07
Branches used to L1 cache (MIPS)	999.977
Percentage of operations per instruction (float)	10.41%
Percentage of operations per instruction (int)	22.10%
MIPS (CPU cycles)	1402.05
MIPS (MIPS)	1401.15
MFLCPS (CPU cycles)	11.5300
MFLCPS (MIPS)	11.411
Processor utilization	46.20%

## 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

- `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

```
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.XXXXXX' (XXXXXX 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

Hpmviz
File Edit

Label	Excl. Dur. (s)	Incl. Dur. (s)	Count
swin			
-Loop 200	89897	89897	1000
-Loop 100	87008	87008	1000
-Loop 300	87007	87007	1000
-Calc2	2.769	60.791	1000
-loop 1%	1.100	1.100	1000
-MPI Calc2	1.004	1.004	1000
-Calc1	0.580	85.483	1000
-MPI Calc1 end	0.730	0.730	1000
-MPI Calc2 end	0.000	0.000	1000
-init0	0.200	0.200	1
-Calc0	0.191	69.467	1000
-Calc0	0.00	0.00	1
-MPI Calc0 start	0.007	0.007	1000
-MPI Calc0 end	0.000	0.000	1000

MPI Calc2 end Metrics	
Task	Count
Thread	20
Count	1000
ErrSec	0.000
ErrDec	0.000
LEPIL Miss	159000
Total LS	4828
Total LS	1.760
MIPS	116281
Total Cycles	0.040
HW HW Cycle	0
FPI+Fwa	0.000
MIPS	0.000
FMA %	0
Cross Int.	0

```

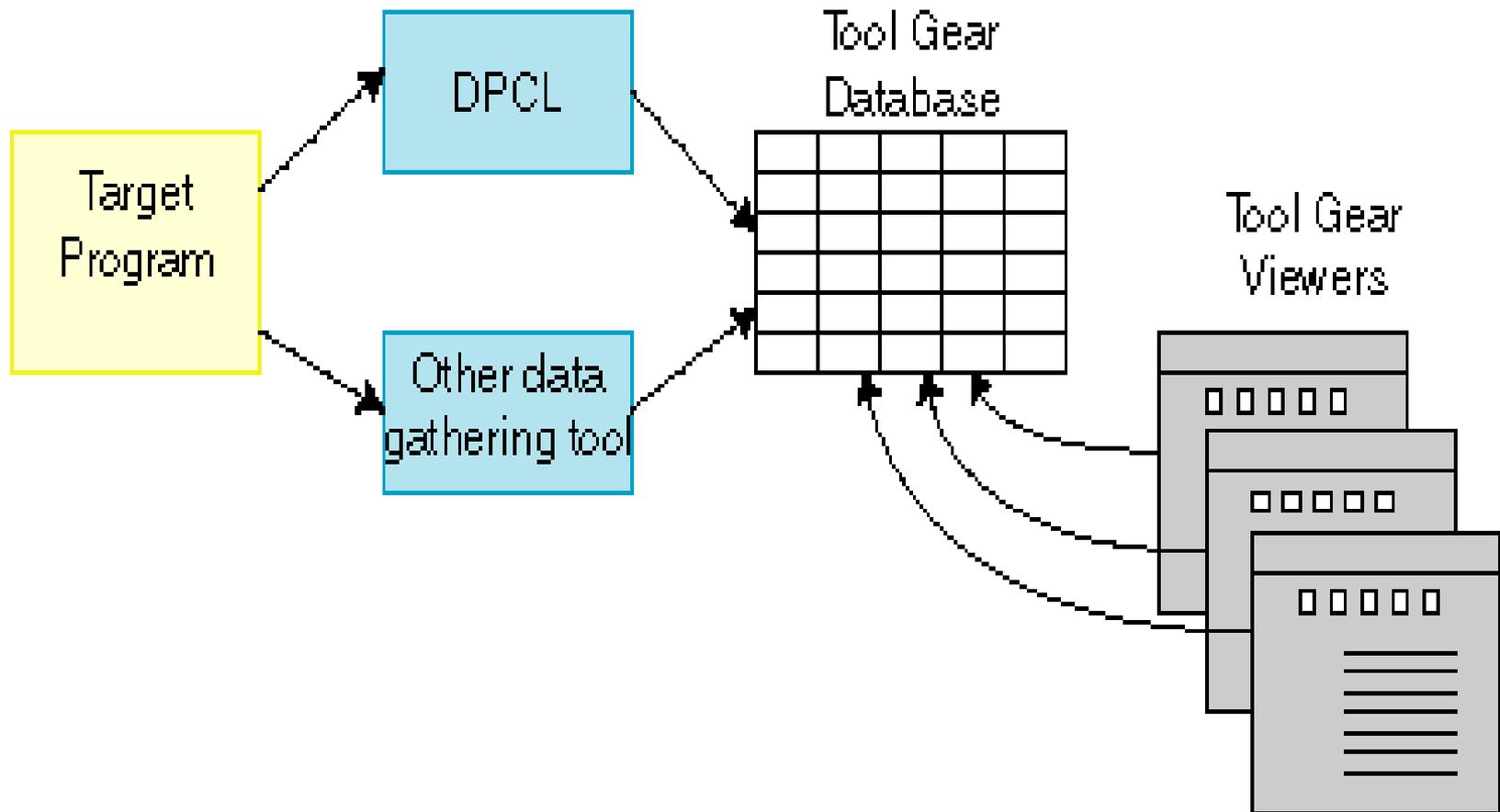
do: f_pprocs; do: "MPI Calc2 end"
if(taskid.eq.0)then
  call mpi_send(VIEW(1,1),n1,MPI_DOUBLE_PRECISION,
  i nrank0-1,1,MPI_COMM_WORLD,req(1),ierr)
endif
if(taskid.eq.nrank0-1)then
  call mpi_recv(VIEW(1,n+1),n1,MPI_DOUBLE_PRECISION,
  i 0,2,MPI_COMM_WORLD,req(2),ierr)
  call mpi_send(VIEW(1,n+1),n1,MPI_DOUBLE_PRECISION,
  i 0,3,MPI_COMM_WORLD,req(3),ierr)
endif
if(taskid.eq.nrank0-1)then
  call mpi_send(VIEW(1,n+1),n1,MPI_DOUBLE_PRECISION,
  i 0,1,MPI_COMM_WORLD,req(4),ierr)
endif
if(taskid.eq.0)then
  call mpi_send(VIEW(1,1),n1,MPI_DOUBLE_PRECISION,
  i nrank0-1,2,MPI_COMM_WORLD,req(5),ierr)
  call mpi_send(VIEW(1,1),n1,MPI_DOUBLE_PRECISION,
  i nrank0-1,3,MPI_COMM_WORLD,req(6),ierr)
endif
if(taskid.eq.0.or.taskid.eq.nrank0-1)then
  call MPI_WAITALL(6,req,stat,ierr)
endif
do 210 i=1,n
  VIEW(i,n+1) = VIEW(i,n)
  VIEW(i,1) = VIEW(i,n+1)
  VIEW(i,n+1) = VIEW(i,1)
enddo 210 CONTINUE
do 220 i=1,n+1
  VIEW(i,n+1) = VIEW(i,n)
  VIEW(i,n) = VIEW(i,n+1)
  VIEW(i,1) = VIEW(i,1)
enddo 220 CONTINUE
if(taskid.eq.nrank0-1)then
  call mpi_send(VIEW(1,n+1),n1,MPI_DOUBLE_PRECISION,
  i 0,2,MPI_COMM_WORLD,req(1),ierr)
  call mpi_send(VIEW(1,n+1),n1,MPI_DOUBLE_PRECISION,
  i 0,3,MPI_COMM_WORLD,req(2),ierr)
endif
if(taskid.eq.0)then
  call mpi_send(VIEW(1,1),n1,MPI_DOUBLE_PRECISION,
  i nrank0-1,2,MPI_COMM_WORLD,req(3),ierr)
  call mpi_send(VIEW(1,1),n1,MPI_DOUBLE_PRECISION,
  i nrank0-1,3,MPI_COMM_WORLD,req(4),ierr)
endif

```

```
#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 );
```

- 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](http://www.llnl.gov/CASC/tool_gear)

# ToolGear Architecture





# ToolGear Screenshot: Instrumentation

1: testcpmod\_mpi on snow (Live)

Run Program has terminated

Line(s)	Source	L1 util	FP Ins	FMA's	Total flops	FLOP/sec
38:	for( i = 0; i < 10; i++ ) {					
39:	/* Tiled */					
40:	[[init_array]]();					
41:						
42:	[[priabs]]("Doing %d flops of tiled test\n", FLOPS);					
43:	[[do_tiled_cache_test]](FLOPS);	0.991517	1.04056e+07	1.4050e+07	2.05716e+07	1.46402e+08
44:						
45:	/* Untiled */					
46:	[[init_array]]();					
47:						
48:	[[priabs]]("Doing %d flops of untiled test\n", FLOPS);					
49:	[[do_untiled_cache_test]](FLOPS);	0.937468	1.04056e+07	1.4050e+07	2.05716e+07	1.43055e+08
50:						
51:	/* Indexed */					
52:	[[init_array]]();					
53:						
54:	[[priabs]]("Doing %d flops of indirect address test\n", FLOPS);					
55:	[[do_indirect_address_test]](FLOPS);	0.932943	1.04056e+07	1.4050e+07	2.05722e+07	3.26941e+07
56:	[[priabs]]("Done with series %i\n", i);					
57:	[[fprintf]](stderr, "Done with series %i\n", i);					

8 data pts: Max 0.93747 (Rank 0/Thread 1) Min 0.937464 (7/1) Mean 0.937468 StdDev 1.90709e-06 Sum 7.49974

Mean L49



# ToolGear Screenshot 2: Tree View

The screenshot shows the ToolGear IDE interface. The title bar reads "1: unt98 on frost (Live)". Below the title bar is a "Run" button and a search icon. The main window is divided into two panes. The left pane shows a tree view of the project structure, with "doBlocks.f" selected. The right pane shows the source code of "doBlocks.f".

```
Line# Source Time (sec) L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25 L26 L27 L28 L29 L30 L31 L32 L33 L34 L35 L36 L37 L38 L39 L40 L41 L42 L43 L44 L45 L46 L47 L48 L49 L50 L51 L52 L53 L54 L55 L56 L57 L58 L59 L60 L61 L62 L63 L64 L65 L66 L67 L68 L69 L70 L71 L72 L73 L74 L75 L76 L77 L78 L79 L80 L81 L82 L83 L84 L85 L86 L87 L88 L89 L90 L91 L92 L93 L94 L95 L96 L97 L98 L99 L100 L101 L102 L103 L104 L105 L106 L107 L108 L109 L110 L111 L112 L113 L114 L115 L116 L117 L118 L119 L120 L121 L122 L123 L124 L125 L126 L127 L128 L129 L130 L131 L132 L133 L134 L135 L136 L137 L138 L139 L140 L141 L142 L143 L144 L145 L146 L147 L148 L149 L150 L151 L152 L153 L154 L155 L156 L157 L158 L159 L160 L161 L162 L163 L164 L165 L166 L167 L168 L169 L170 L171 L172 L173 L174 L175 L176 L177 L178 L179 L180 L181 L182 L183 L184 L185 L186 L187 L188 L189 L190 L191 L192 L193 L194 L195 L196 L197 L198 L199 L200 L201 L202 L203 L204 L205 L206 L207 L208 L209 L210 L211 L212 L213 L214 L215 L216 L217 L218 L219 L220 L221 L222 L223 L224 L225 L226 L227 L228 L229 L230 L231 L232 L233 L234 L235 L236 L237 L238 L239 L240 L241 L242 L243 L244 L245 L246 L247 L248 L249 L250 L251 L252 L253 L254 L255 L256 L257 L258 L259 L260 L261 L262 L263 L264 L265 L266 L267 L268 L269 L270 L271 L272 L273 L274 L275 L276 L277 L278 L279 L280 L281 L282 L283 L284 L285 L286 L287 L288 L289 L290 L291 L292 L293 L294 L295 L296 L297 L298 L299 L300 L301 L302 L303 L304 L305 L306 L307 L308 L309 L310 L311 L312 L313 L314 L315 L316 L317 L318 L319 L320 L321 L322 L323 L324 L325 L326 L327 L328 L329 L330 L331 L332 L333 L334 L335 L336 L337 L338 L339 L340 L341 L342 L343 L344 L345 L346 L347 L348 L349 L350 L351 L352 L353 L354 L355 L356 L357 L358 L359 L360 L361 L362 L363 L364 L365 L366 L367 L368 L369 L370 L371 L372 L373 L374 L375 L376 L377 L378 L379 L380 L381 L382 L383 L384 L385 L386 L387 L388 L389 L390 L391 L392 L393 L394 L395 L396 L397 L398 L399 L400 L401 L402 L403 L404 L405 L406 L407 L408 L409 L410 L411 L412 L413 L414 L415 L416 L417 L418 L419 L420 L421 L422 L423 L424 L425 L426 L427 L428 L429 L430 L431 L432 L433 L434 L435 L436 L437 L438 L439 L440 L441 L442 L443 L444 L445 L446 L447 L448 L449 L450 L451 L452 L453 L454 L455 L456 L457 L458 L459 L460 L461 L462 L463 L464 L465 L466 L467 L468 L469 L470 L471 L472 L473 L474 L475 L476 L477 L478 L479 L480 L481 L482 L483 L484 L485 L486 L487 L488 L489 L490 L491 L492 L493 L494 L495 L496 L497 L498 L499 L500
```

Source: unt98 on frost (Live)

- unt98 on frost (Live)
  - LIBC 1
  - lib 1
  - math 1
  - assert 1
  - assert\_yosh 1
  - pool 1
  - constant\_pool 1
  - doBlocks 1
  - doBlocks.f
  - doBlocks
  - rand 1

```
integer4 random (range)
```

-----

- Using a random number generator which generates uniform random
- integers in the range 0 ≤ r < n, generate a random number
- in the range 0 ≤ r < range where range is a full base integer

-----

```
implicit integer4 (1-2)  
integer4 range  
real8 rand  
  
||| rand = floor( |rand| / (0 + range))
```

- Handle the rare case where the floating point random 1 is exactly 1

```
if( rand == (range + 1)) rand = range
```

doBlocks.f

# ToolGear Screenshot 3: MPI Profiling

1: pi3 on snow (Live)

Run  Program has terminated

Line(s)	Source	Count	Max time	Mintime	Mean time <sup>4</sup>	App%	MPI%
	pi3 on snow (Live)	16	6.57	0.518	2.541	23.41	100
	pi3.f	16	6.57	0.518	2.541	23.41	100
( 1- 71):	main	16	6.57	0.518	2.541	23.41	100
43:	call MPI_ICAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)	8	2.19	0.228	0.611	5.63	24.06
55:	call MPI_REDUCE(mypa,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,	8	4.38	0.29	1.93	17.78	75.94
1:	c pi3.f						
2:	c slightly modified from the MPICE pi3 example code						
3:	c*****						
4:	c pi.f - compute pi by integrating $f(x) = 4/(1+x^2)$						
5:	c						

pi3.f main l2

- 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

- 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
- Full Loop, CFG and Basic Block decoding

- 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/>

- 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!

- 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
```

- 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.

- 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

- 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:  $T_{child} = \text{Inclusive } T_{function}$

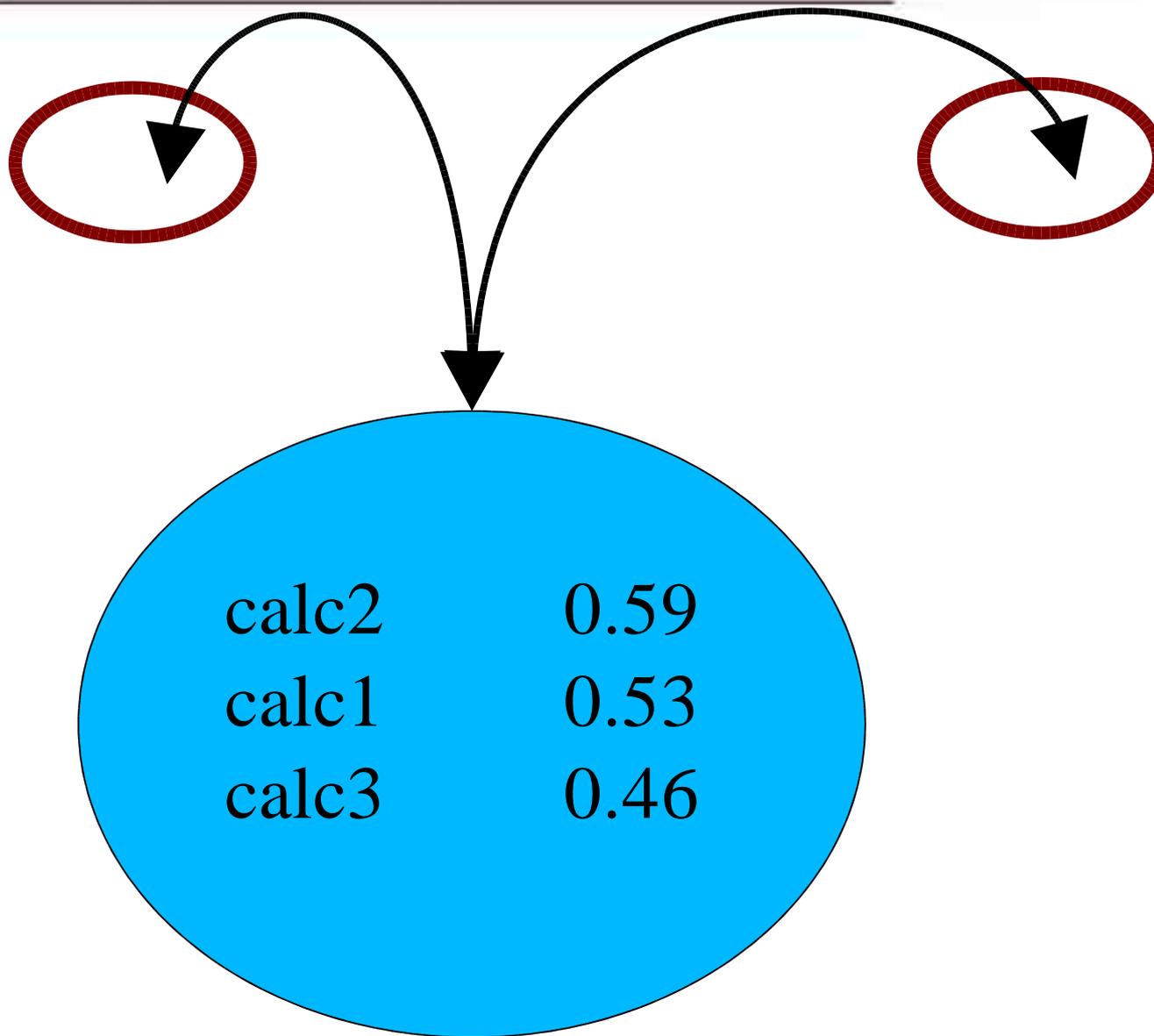
```
(dynaprof) use probes/papiprobe PAPI_TOT_CYC, PAPI_TOT_INS
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr function swim.F calc*
Swim.F, inserted 3 instrumentation points
(dynaprof) instr
calc1_
calc2_
calc3_
calc3z_
```



# Swim Benchmark: Cycles & Instructions

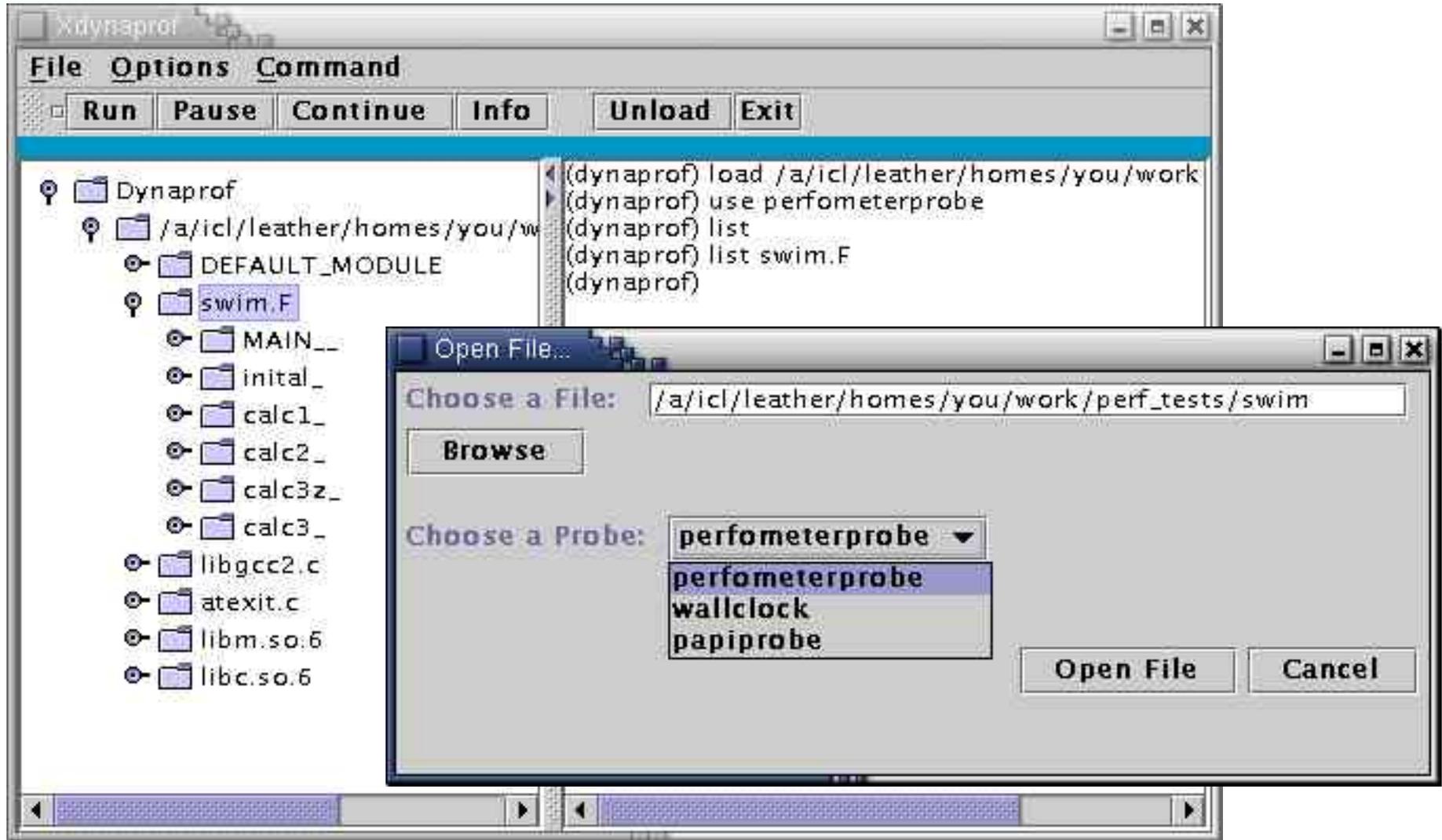
---

# Swim Benchmark: Instructions per Cycle



- 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)
- Does not work well!

# DynaProf GUI Screenshot



- It's a bit rough
- Supported Platforms
  - Using DynInst 3.0
    - Linux 2.x
    - AIX 4.3/5?
    - Solaris 2.8
    - IRIX 6.x
  - Using DPCL (formal MPI support)
    - AIX 4.3
    - AIX 5
- Available as a development snapshot from:
- Includes:
  - Java/Swing GUI
  - User's Guide
  - Probe libraries

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

- Port to DynInst API 4.0 (Released RSN)
- IA64 Support
- New instrumentation point support:
  - Object
  - Instance
  - Loop
  - Basic Block
  - Arbitrary
- Breakpoints
- Support for programs that dynamically load modules during run-time. (Mozilla)
- Integration with TAU

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

- 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 and back to the user? Maybe...
  - Tool Daemon Protocol: U. Wisc
  - DPCL for all DynInst: LANL and me
- How do we correlate performance data from optimized code to the source?
- We want to understand all aspects of a program's performance. What about behaviour over time?

- Statistical profiling is often static
  - Gprof, Quantify, Speedshop, Workshop, Tprof, etc...
- Applications vs. kernels have distinct phases.
  - Initialization
  - Data input
    - Compute
    - Communicate
    - Repeat
  - Data output
  - Finalization

- Workloads on the hardware are most often periodic.
- More 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?