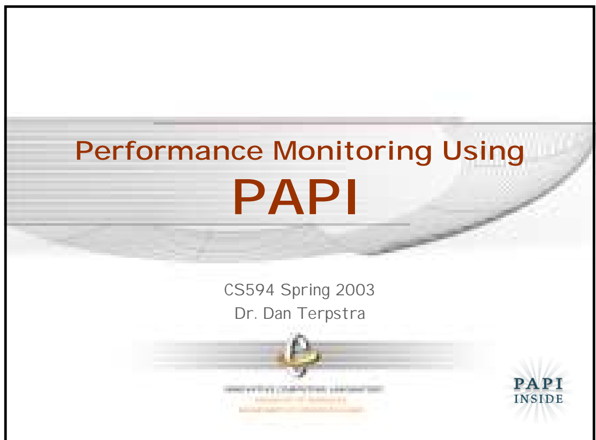
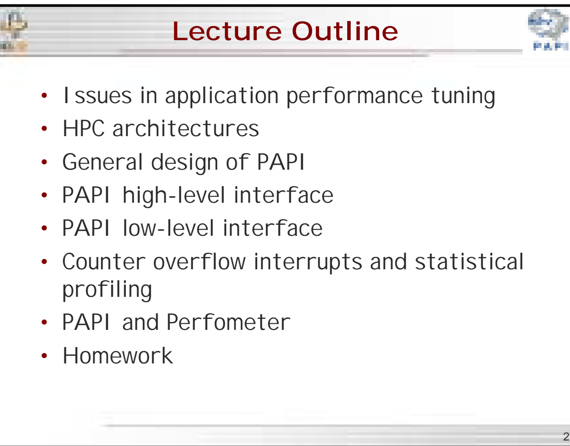
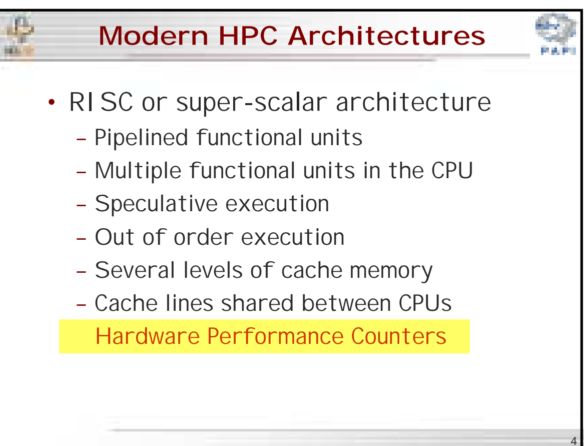
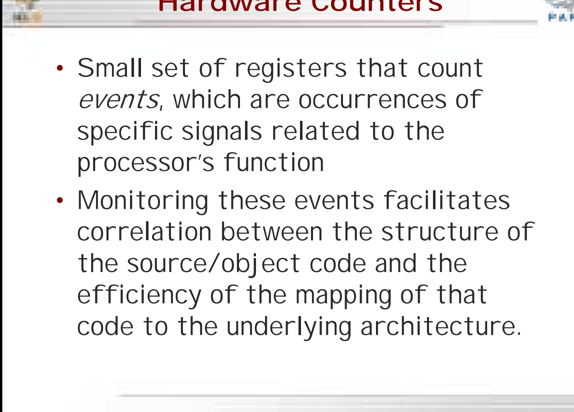
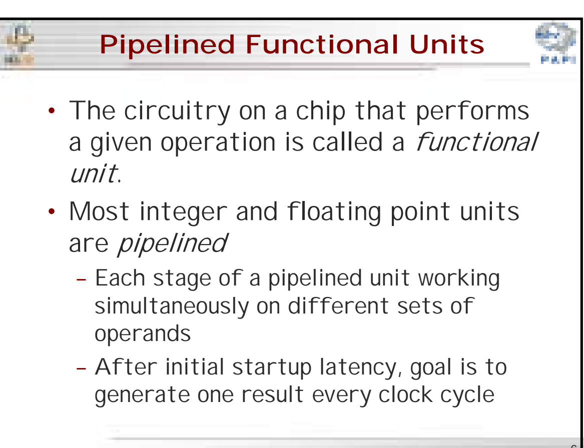

Performance Monitoring Using
PAPI
CS594 Spring 2003
Dr. Dan Terpstra
PAPI
INSIDE

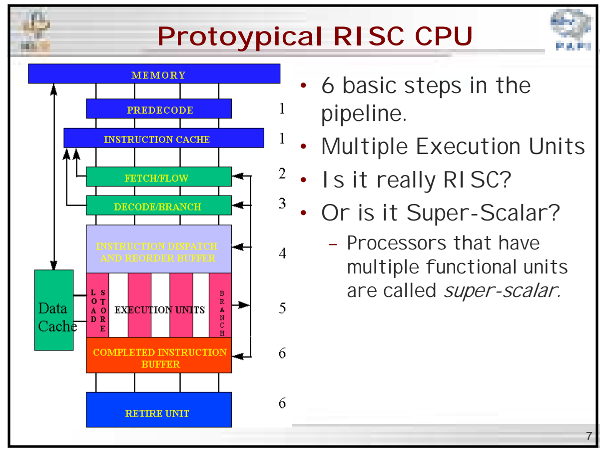

Lecture Outline
• Issues in application performance tuning
• HPC architectures
• General design of PAPI
• PAPI high-level interface
• PAPI low-level interface
• Counter overflow interrupts and statistical profiling
• PAPI and Perfometer
• Homework
2

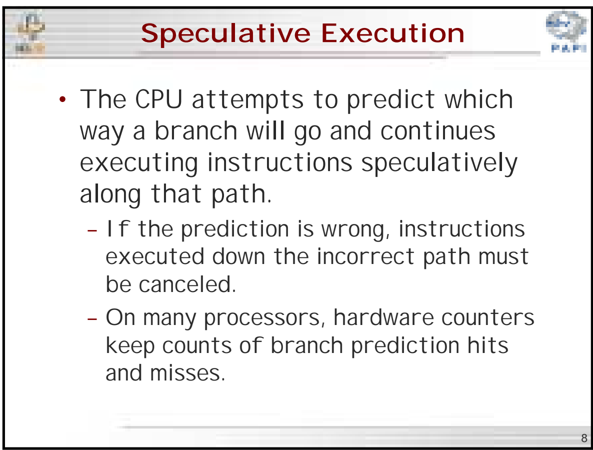
Application Performance Tuning on Modern High Performance Computing Architectures

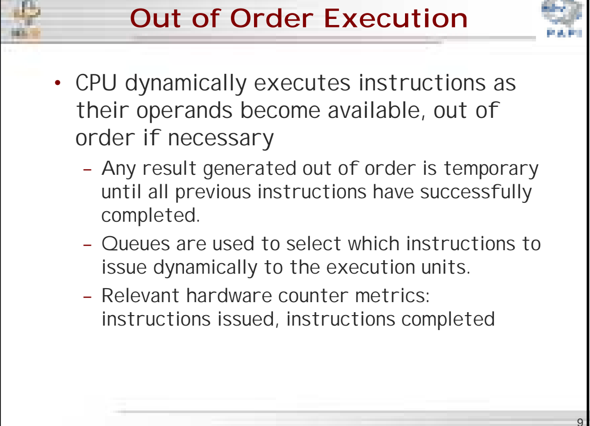

Modern HPC Architectures
• RISC or super-scalar architecture
– Pipelined functional units
– Multiple functional units in the CPU
– Speculative execution
– Out of order execution
– Several levels of cache memory
– Cache lines shared between CPUs
Hardware Performance Counters
4

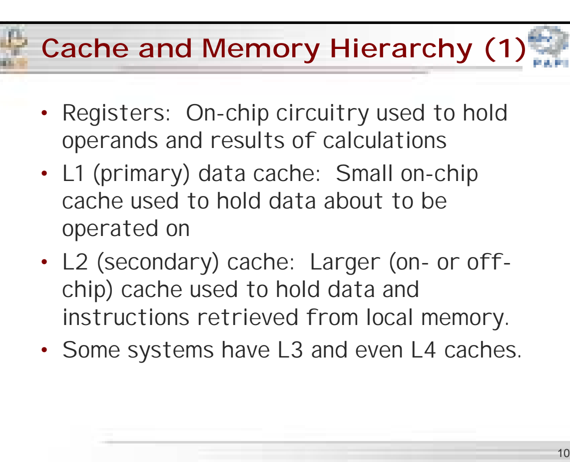

Hardware Counters
• Small set of registers that count events, which are occurrences of specific signals related to the processor's function
• Monitoring these events facilitates correlation between the structure of the source/object code and the efficiency of the mapping of that code to the underlying architecture.
5

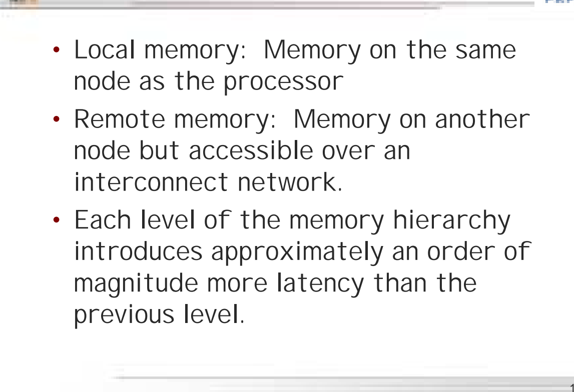

Pipelined Functional Units
• The circuitry on a chip that performs a given operation is called a functional unit.
• Most integer and floating point units are pipelined
– Each stage of a pipelined unit working simultaneously on different sets of operands
– After initial startup latency, goal is to generate one result every clock cycle
6

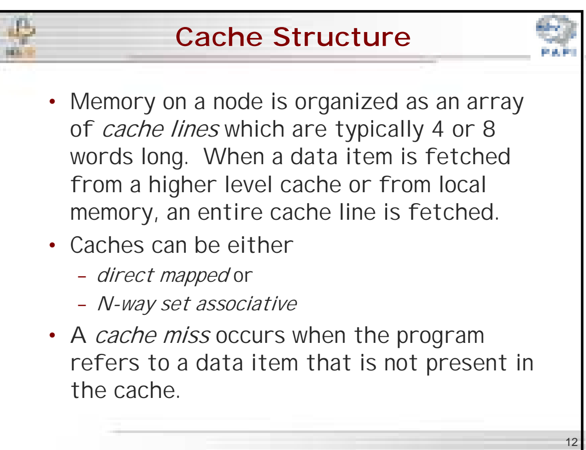

Protoypical RISC CPU
MEMORY
PREDECODE
INSTRUCTION CACHE
FETCH/FLOW
DECODE/BRANCH
INSTRUCTION DISPATCH AND REORDER BUFFER
Data Cache
LOAD
STORE
EXECUTION UNITS
BRANCH
COMPLETED INSTRUCTION BUFFER
RETIRE UNIT
1
1
2
3
4
5
6
6
• 6 basic steps in the pipeline.
• Multiple Execution Units
• Is it really RISC?
• Or is it Super-Scalar?
– Processors that have multiple functional units are called super-scalar.
7

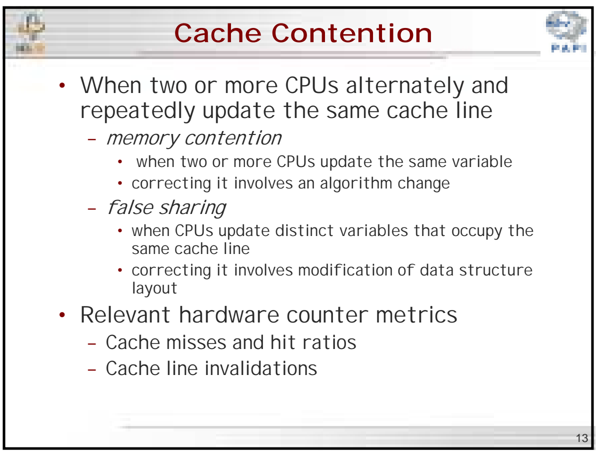

Speculative Execution
• The CPU attempts to predict which way a branch will go and continues executing instructions speculatively along that path.
– If the prediction is wrong, instructions executed down the incorrect path must be canceled.
– On many processors, hardware counters keep counts of branch prediction hits and misses.
8

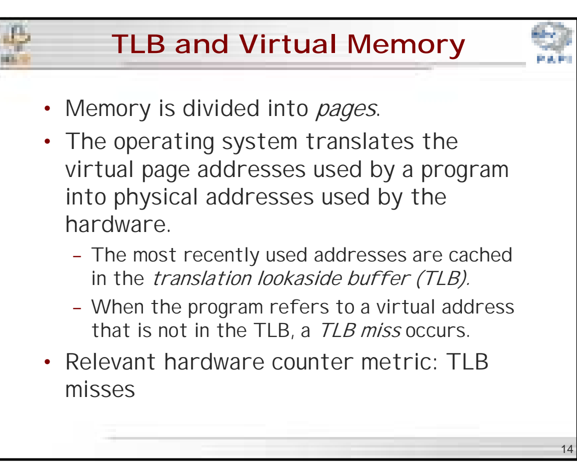

Out of Order Execution
• CPU dynamically executes instructions as their operands become available, out of order if necessary
– Any result generated out of order is temporary until all previous instructions have successfully completed.
– Queues are used to select which instructions to issue dynamically to the execution units.
– Relevant hardware counter metrics: instructions issued, instructions completed
9

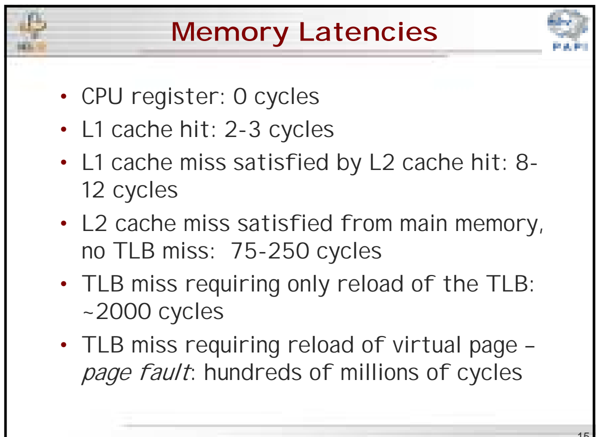

Cache and Memory Hierarchy (1)
• Registers: On-chip circuitry used to hold operands and results of calculations
• L1 (primary) data cache: Small on-chip cache used to hold data about to be operated on
• L2 (secondary) cache: Larger (on- or off-chip) cache used to hold data and instructions retrieved from local memory.
• Some systems have L3 and even L4 caches.
10

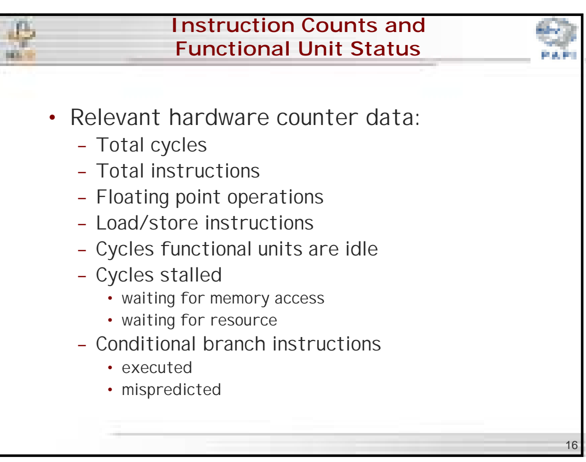

Cache and Memory Hierarchy (2)
• Local memory: Memory on the same node as the processor
• Remote memory: Memory on another node but accessible over an interconnect network.
• Each level of the memory hierarchy introduces approximately an order of magnitude more latency than the previous level.
11

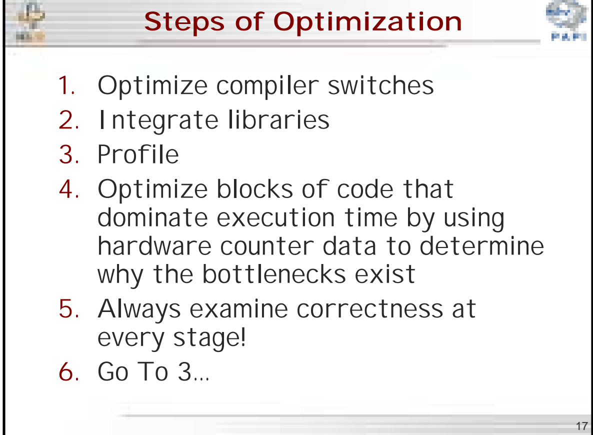

Cache Structure
• Memory on a node is organized as an array of cache lines which are typically 4 or 8 words long. When a data item is fetched from a higher level cache or from local memory, an entire cache line is fetched.
• Caches can be either
– direct mapped or
– N-way set associative
• A cache miss occurs when the program refers to a data item that is not present in the cache.
12

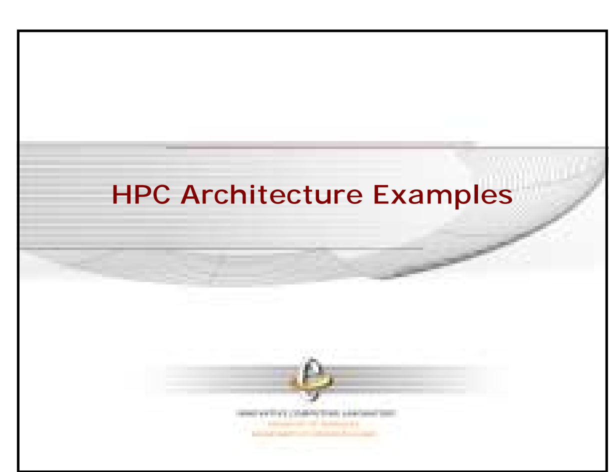

Cache Contention
• When two or more CPUs alternately and repeatedly update the same cache line
– memory contention
• when two or more CPUs update the same variable
• correcting it involves an algorithm change
– false sharing
• when CPUs update distinct variables that occupy the same cache line
• correcting it involves modification of data structure layout
• Relevant hardware counter metrics
– Cache misses and hit ratios
– Cache line invalidations
13

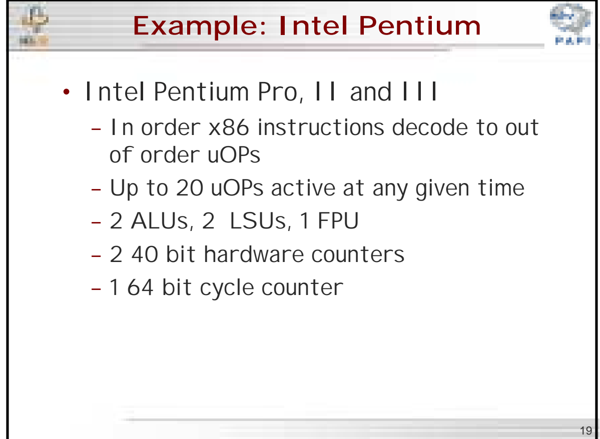

TLB and Virtual Memory
• Memory is divided into pages.
• The operating system translates the virtual page addresses used by a program into physical addresses used by the hardware.
– The most recently used addresses are cached in the translation lookaside buffer (TLB).
– When the program refers to a virtual address that is not in the TLB, a TLB miss occurs.
• Relevant hardware counter metric: TLB misses
14

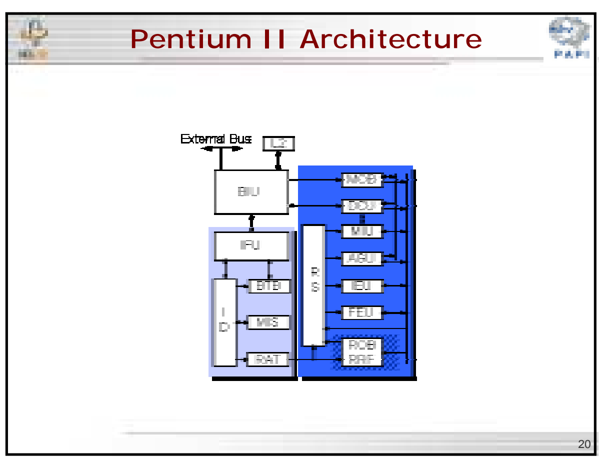

Memory Latencies
• CPU register: 0 cycles
• L1 cache hit: 2-3 cycles
• L1 cache miss satisfied by L2 cache hit: 8-12 cycles
• L2 cache miss satisfied from main memory, no TLB miss: 75-250 cycles
• TLB miss requiring only reload of the TLB: ~2000 cycles
• TLB miss requiring reload of virtual page – page fault: hundreds of millions of cycles
15

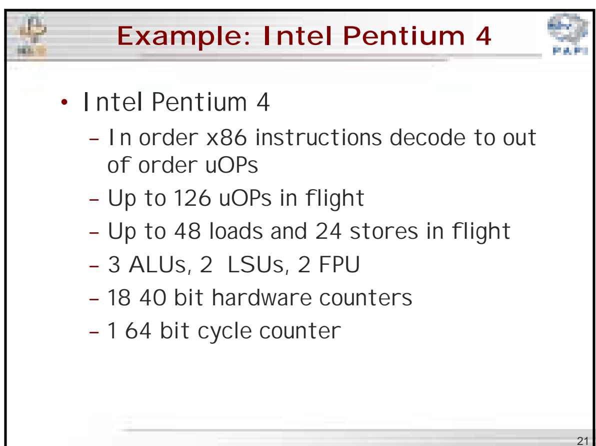

Instruction Counts and Functional Unit Status
• Relevant hardware counter data:
– Total cycles
– Total instructions
– Floating point operations
– Load/store instructions
– Cycles functional units are idle
– Cycles stalled
• waiting for memory access
• waiting for resource
– Conditional branch instructions
• executed
• mispredicted
16

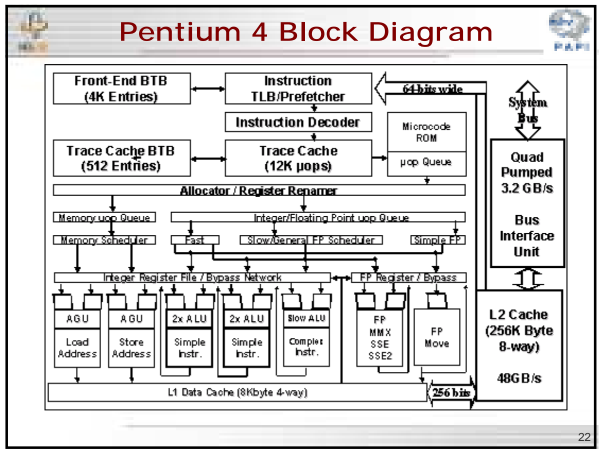

Steps of Optimization
1. Optimize compiler switches
2. Integrate libraries
3. Profile
4. Optimize blocks of code that dominate execution time by using hardware counter data to determine why the bottlenecks exist
5. Always examine correctness at every stage!
6. Go To 3...
17

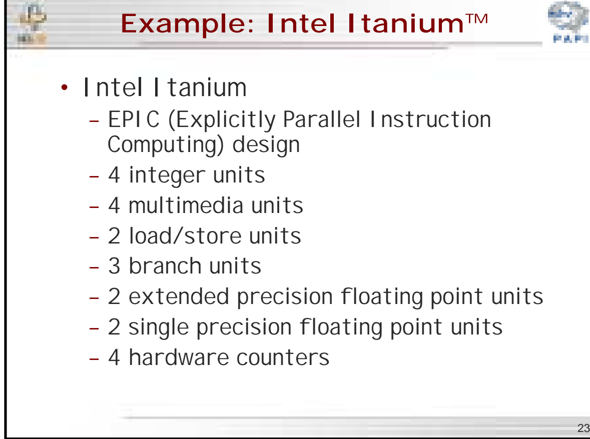

HPC Architecture Examples

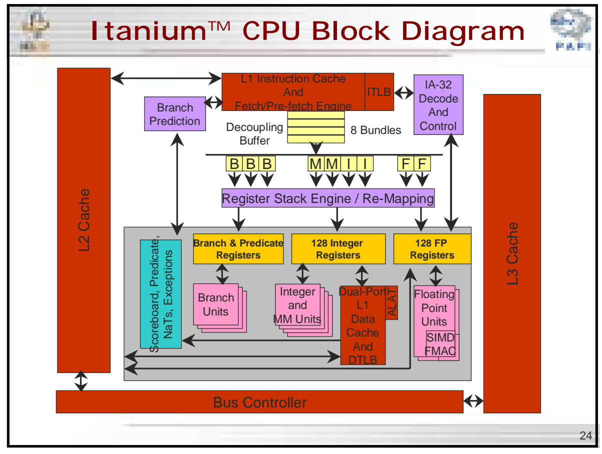

Example: Intel Pentium
• Intel Pentium Pro, II and III
– In order x86 instructions decode to out of order uOPs
– Up to 20 uOPs active at any given time
– 2 ALUs, 2 LSUs, 1 FPU
– 2 40 bit hardware counters
– 1 64 bit cycle counter
19

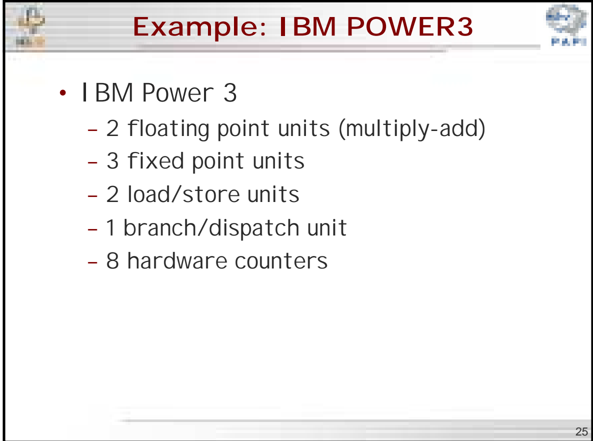

Pentium II Architecture
External Bus
L2
BIU
IFU
BTB
MIS
RAT
RS
DCU
MIU
AGU
IEU
FEU
ROB
RRF
20

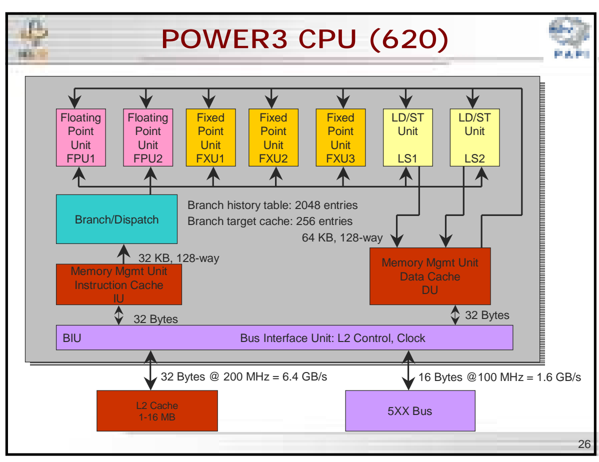

Example: Intel Pentium 4
• Intel Pentium 4
– In order x86 instructions decode to out of order uOPs
– Up to 126 uOPs in flight
– Up to 48 loads and 24 stores in flight
– 3 ALUs, 2 LSUs, 2 FPU
– 18 40 bit hardware counters
– 1 64 bit cycle counter
21

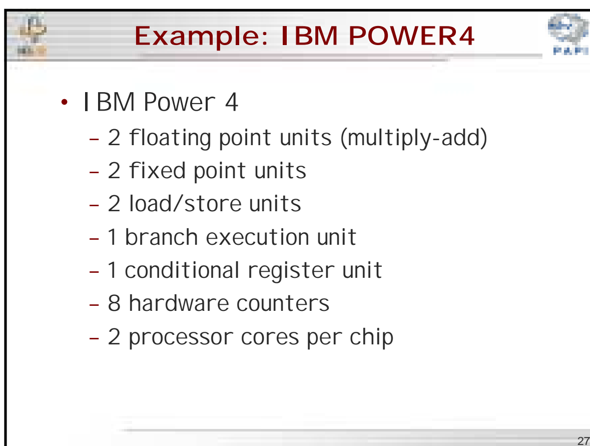

Pentium 4 Block Diagram
Front-End BTB (4K Entries)
Instruction TLB/Prefetcher
64-bits wide
System Bus
Instruction Decoder
Microcode ROM
Trace Cache BTB (512 Entries)
Trace Cache (12K µops)
µop Queue
Quad Pumped 3.2 GB/s
Allocator / Register Renamer
Memory uop Queue
Integer/Floating Point uop Queue
Bus Interface Unit
Memory Scheduler
Fast
Slow/General FP Scheduler
Simple FP
Integer Register File / Bypass Network
FP Register / Bypass
AGU
Load Address
AGU
Store Address
2x ALU
Simple Instr.
2x ALU
Simple Instr.
Slow ALU
Complex Instr.
FP MMX SSE SSE2
FP Move
L2 Cache (256K Byte 8-way)
48GB/s
L1 Data Cache (8Kbyte 4-way)
256 bits
22

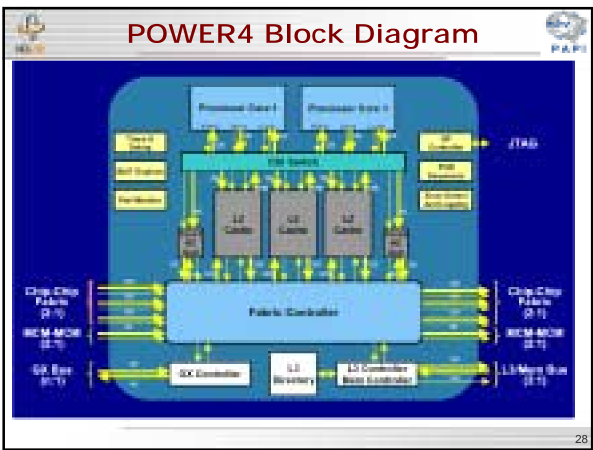

Example: Intel Itanium™
• Intel Itanium
– EPIC (Explicitly Parallel Instruction Computing) design
– 4 integer units
– 4 multimedia units
– 2 load/store units
– 3 branch units
– 2 extended precision floating point units
– 2 single precision floating point units
– 4 hardware counters
23

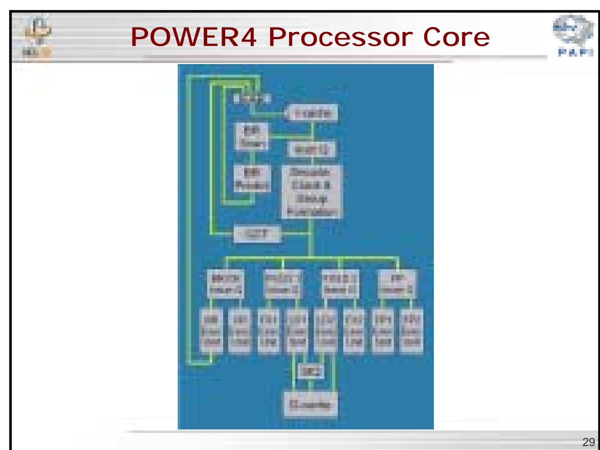

Itanium™ CPU Block Diagram
L2 Cache
L1 Instruction Cache And Fetch/Pre-fetch Engine
ITLB
IA-32 Decode And Control
Branch Prediction
Decoupling Buffer
8 Bundles
B B B
M M I I
F F
Register Stack Engine / Re-Mapping
Scoreboard, Predicate, NaTs, Exceptions
Branch & Predicate Registers
128 Integer Registers
128 FP Registers
Branch Units
Integer and MM Units
Dual-Port L1 Data Cache And DTLB
ALAT
Floating Point Units
SIMD FMAC
L3 Cache
Bus Controller
24

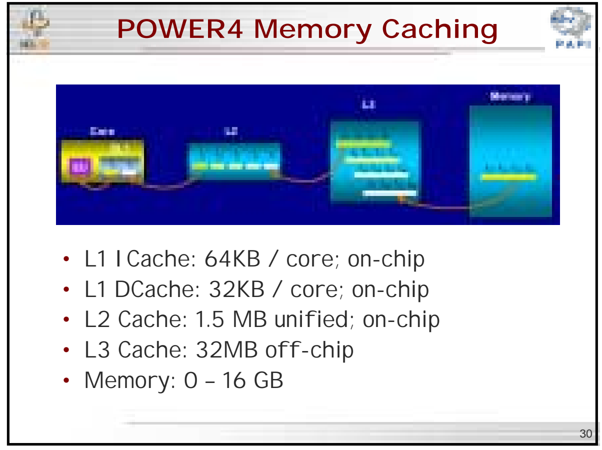

Example: IBM POWER3
• IBM Power 3
– 2 floating point units (multiply-add)
– 3 fixed point units
– 2 load/store units
– 1 branch/dispatch unit
– 8 hardware counters
25

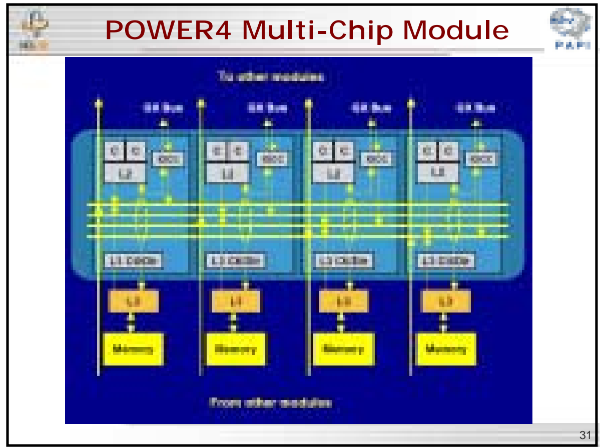

POWER3 CPU (620)
Floating Point Unit FPU1
Floating Point Unit FPU2
Fixed Point Unit FXU1
Fixed Point Unit FXU2
Fixed Point Unit FXU3
LD/ST Unit LS1
LD/ST Unit LS2
Branch/Dispatch
Branch history table: 2048 entries
Branch target cache: 256 entries
64 KB, 128-way
32 KB, 128-way
Memory Mgmt Unit Instruction Cache IU
Memory Mgmt Unit Data Cache DU
32 Bytes
32 Bytes
BIU
Bus Interface Unit: L2 Control, Clock
32 Bytes @ 200 MHz = 6.4 GB/s
16 Bytes @100 MHz = 1.6 GB/s
L2 Cache 1-16 MB
5XX Bus
26

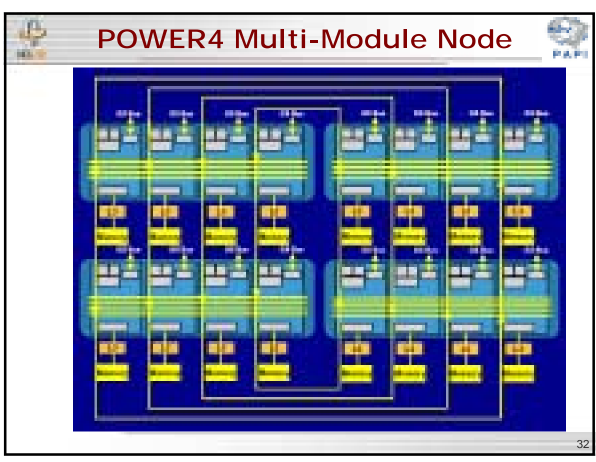

Example: IBM POWER4
• IBM Power 4
– 2 floating point units (multiply-add)
– 2 fixed point units
– 2 load/store units
– 1 branch execution unit
– 1 conditional register unit
– 8 hardware counters
– 2 processor cores per chip
27


POWER4 Block Diagram
28

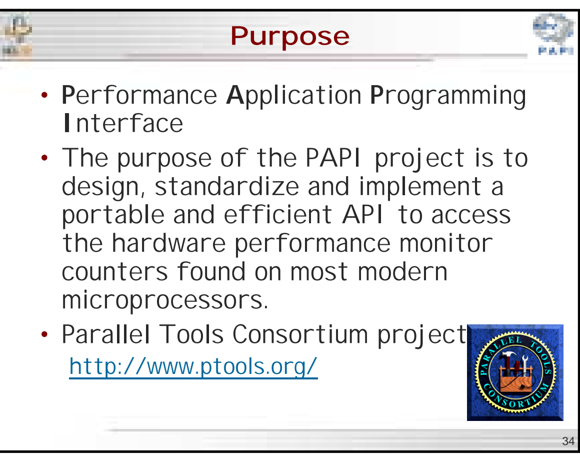

POWER4 Processor Core
29

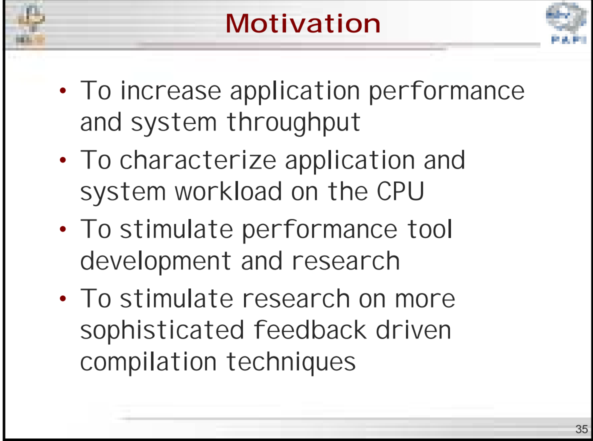

POWER4 Memory Caching
• L1 I Cache: 64KB / core; on-chip
• L1 DCache: 32KB / core; on-chip
• L2 Cache: 1.5 MB unified; on-chip
• L3 Cache: 32MB off-chip
• Memory: 0 – 16 GB
30

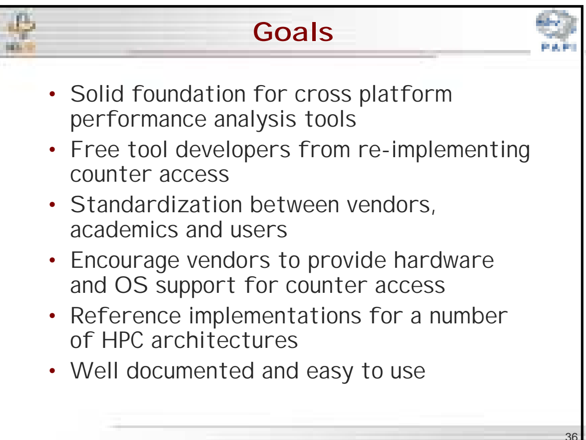

POWER4 Multi-Chip Module
To other modules
From other modules
31


POWER4 Multi-Module Node
32

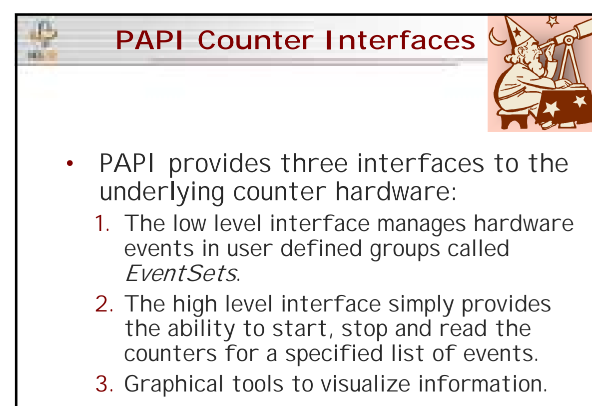
PAPI
PAPI
The
Performance Application Programming Interface

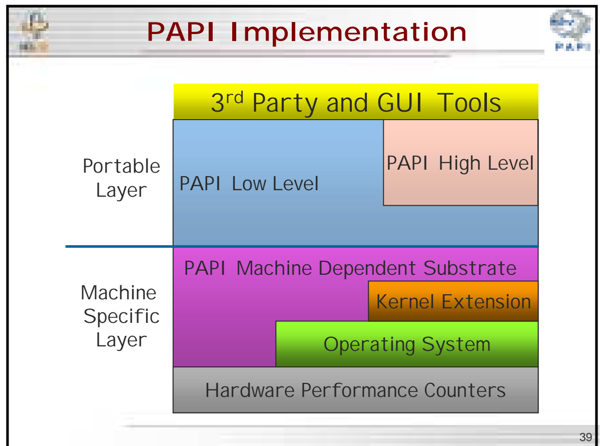

Purpose
• Performance Application Programming Interface
• The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
• Parallel Tools Consortium project
http://www.ptools.org/
34

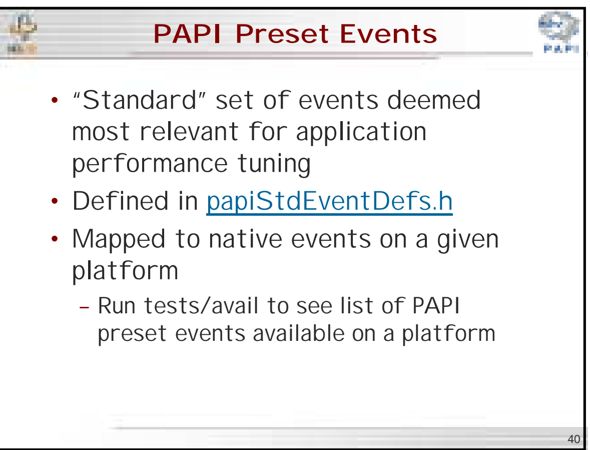

Motivation
• To increase application performance and system throughput
• To characterize application and system workload on the CPU
• To stimulate performance tool development and research
• To stimulate research on more sophisticated feedback driven compilation techniques
35

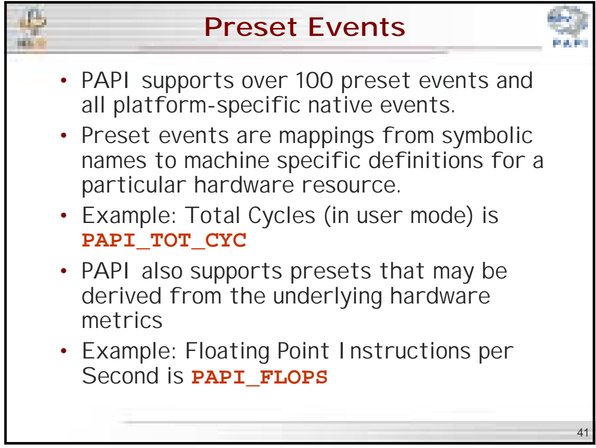

Goals
• Solid foundation for cross platform performance analysis tools
• Free tool developers from re-implementing counter access
• Standardization between vendors, academics and users
• Encourage vendors to provide hardware and OS support for counter access
• Reference implementations for a number of HPC architectures
• Well documented and easy to use
36

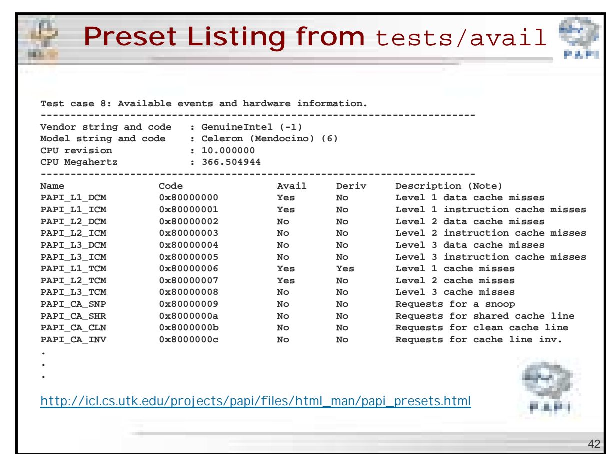
General Design of PAPI

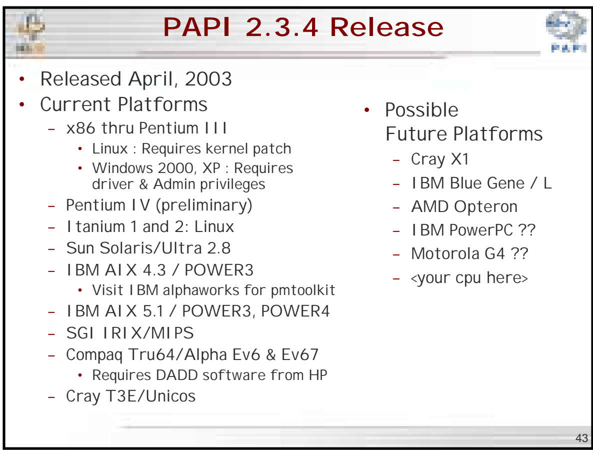

PAPI Counter Interfaces
• PAPI provides three interfaces to the underlying counter hardware:
1. The low level interface manages hardware events in user defined groups called EventSets.
2. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
3. Graphical tools to visualize information.
38

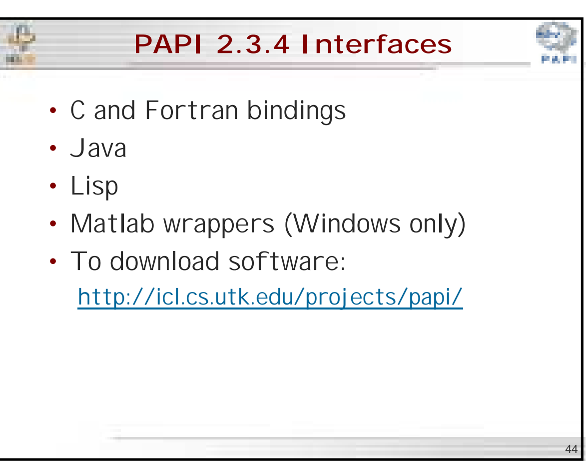

PAPI Implementation
3rd Party and GUI Tools
Portable Layer
PAPI Low Level
PAPI High Level
Machine Specific Layer
PAPI Machine Dependent Substrate
Kernel Extension
Operating System
Hardware Performance Counters
39


PAPI Preset Events
• "Standard" set of events deemed most relevant for application performance tuning
• Defined in papiStdEventDefs.h
• Mapped to native events on a given platform
– Run tests/avail to see list of PAPI preset events available on a platform
40

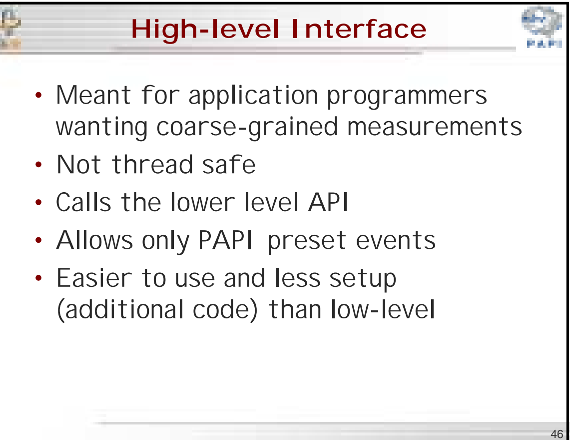

Preset Events
• PAPI supports over 100 preset events and all platform-specific native events.
• Preset events are mappings from symbolic names to machine specific definitions for a particular hardware resource.
• Example: Total Cycles (in user mode) is PAPI_TOT_CYC
• PAPI also supports presets that may be derived from the underlying hardware metrics
• Example: Floating Point Instructions per Second is PAPI_FLOPS
41

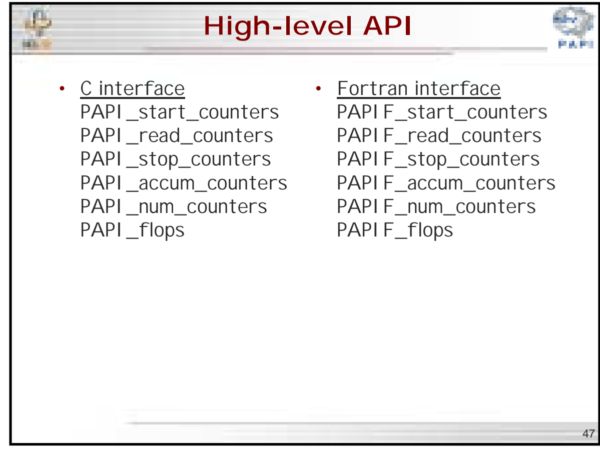

Preset Listing from tests/avail
Test case 8: Available events and hardware information.
Vendor string and code : GenuineIntel (-1)
Model string and code : Celeron (Mendocino) (6)
CPU revision : 10.000000
CPU Megahertz : 366.504944
Name Code Avail Deriv Description (Note)
PAPI_L1_DCM 0x80000000 Yes No 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.
http://icl.cs.utk.edu/projects/papi/files/html_man/papi_presets.html
42

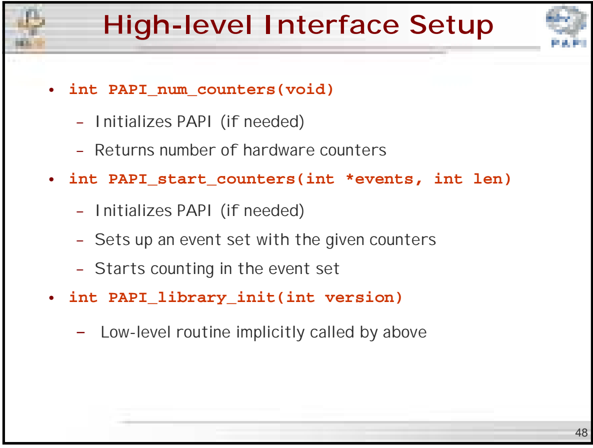

PAPI 2.3.4 Release
• Released April, 2003
• Current Platforms
– x86 thru Pentium III
• Linux : Requires kernel patch
• Windows 2000, XP : Requires driver & Admin privileges
– Pentium IV (preliminary)
– Itanium 1 and 2: Linux
– Sun Solaris/Ultra 2.8
– IBM AIX 4.3 / POWER3
• Visit IBM alphaworks for pmtoolkit
– IBM AIX 5.1 / POWER3, POWER4
– SGI IRIX/MIPS
– Compaq Tru64/Alpha Ev6 & Ev67
• Requires DADD software from HP
– Cray T3E/Unicos
• Possible Future Platforms
– Cray X1
– IBM Blue Gene / L
– AMD Opteron
– IBM PowerPC ??
– Motorola G4 ??
– <your cpu here>
43

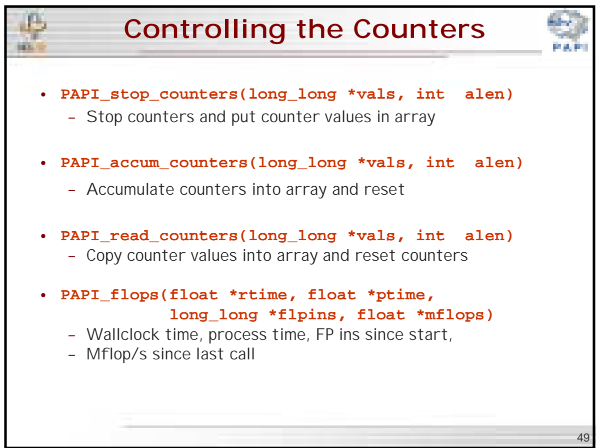

PAPI 2.3.4 Interfaces
• C and Fortran bindings
• Java
• Lisp
• Matlab wrappers (Windows only)
• To download software:
http://icl.cs.utk.edu/projects/papi/
44

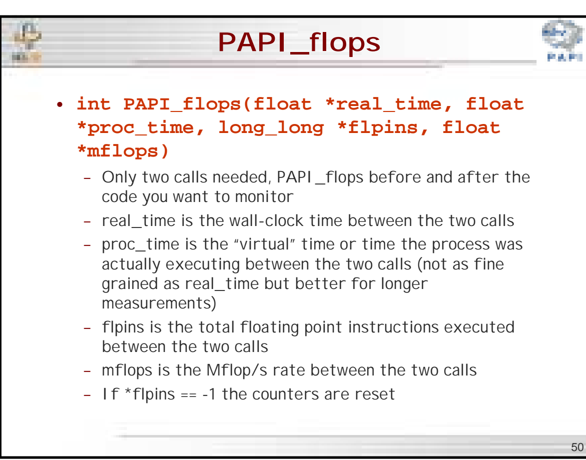
PAPI High-Level Interface

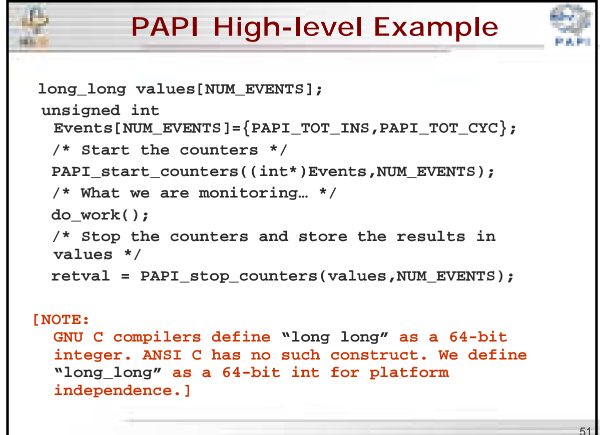

High-level Interface
• Meant for application programmers wanting coarse-grained measurements
• Not thread safe
• Calls the lower level API
• Allows only PAPI preset events
• Easier to use and less setup (additional code) than low-level
46

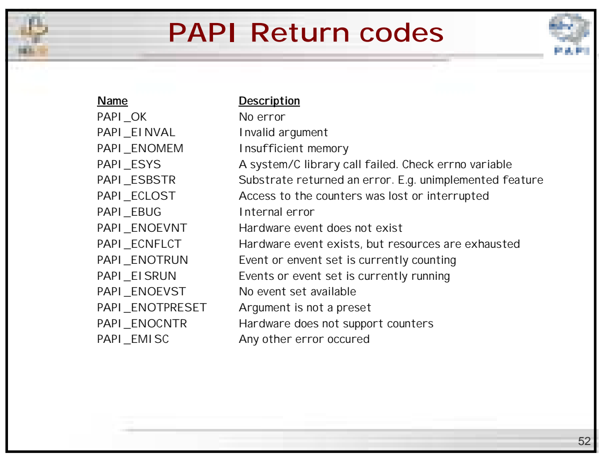

High-level API
• C interface
PAPI_start_counters
PAPI_read_counters
PAPI_stop_counters
PAPI_accum_counters
PAPI_num_counters
PAPI_flops
• Fortran interface
PAPIF_start_counters
PAPIF_read_counters
PAPIF_stop_counters
PAPIF_accum_counters
PAPIF_num_counters
PAPIF_flops
47

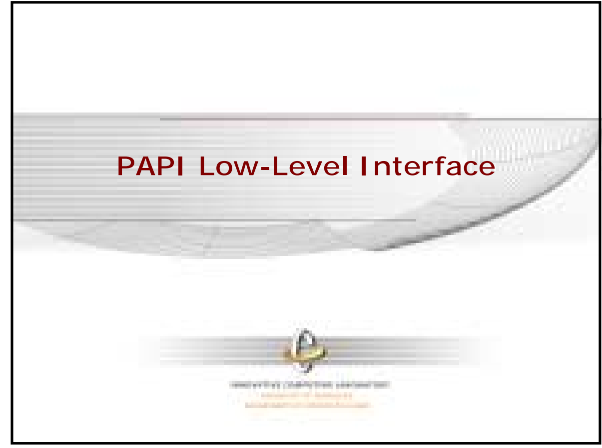

High-level Interface Setup
• int PAPI_num_counters(void)
– Initializes PAPI (if needed)
– Returns number of hardware counters
• int PAPI_start_counters(int *events, int len)
– Initializes PAPI (if needed)
– Sets up an event set with the given counters
– Starts counting in the event set
• int PAPI_library_init(int version)
– Low-level routine implicitly called by above
48

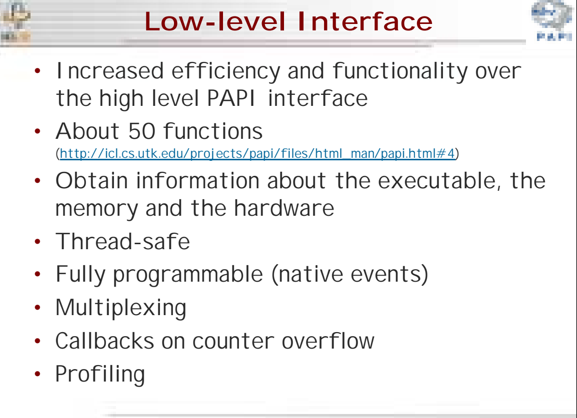

Controlling the Counters
• PAPI_stop_counters(long_long *vals, int alen)
– Stop counters and put counter values in array
• PAPI_accum_counters(long_long *vals, int alen)
– Accumulate counters into array and reset
• PAPI_read_counters(long_long *vals, int alen)
– Copy counter values into array and reset 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
49

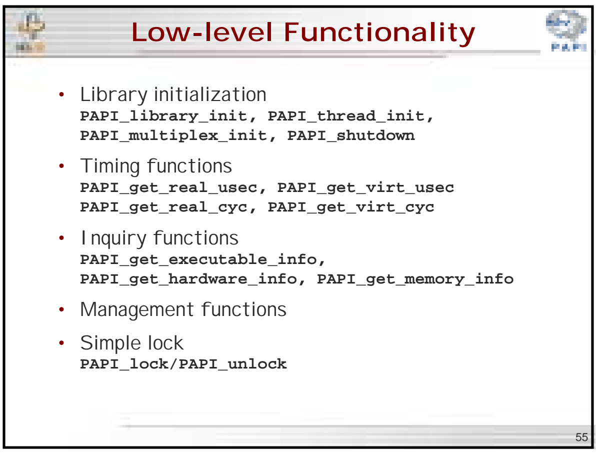

PAPI_flops
• int PAPI_flops(float *real_time, float *proc_time, long_long *flpins, float *mflops)
– Only two calls needed, PAPI_flops before and after the code you want to monitor
– real_time is the wall-clock time between the two calls
– proc_time is the "virtual" time or time the process was actually executing between the two calls (not as fine grained as real_time but better for longer measurements)
– flpins is the total floating point instructions executed between the two calls
– mflops is the Mflop/s rate between the two calls
– If *flpins == -1 the counters are reset
50

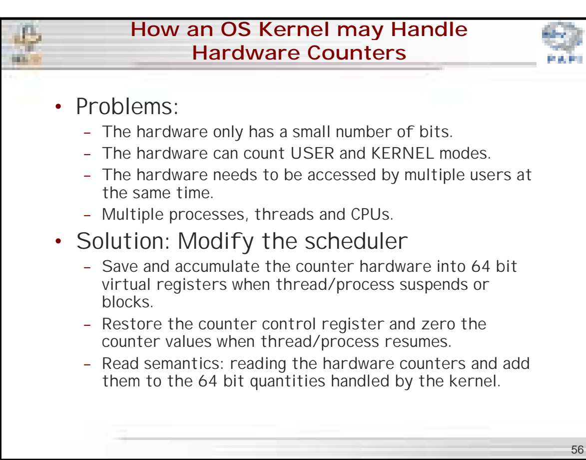

PAPI High-level Example
long_long values[NUM_EVENTS];
unsigned int
Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};
/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);
/* What we are monitoring… */
do_work();
/* Stop the counters and store the results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
[NOTE:
GNU C compilers define "long long" as a 64-bit integer. ANSI C has no such construct. We define "long_long" as a 64-bit int for platform independence.]
51

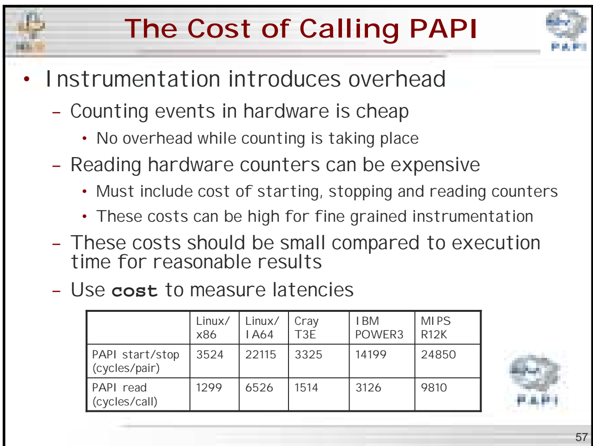

PAPI Return codes
Name
Description
PAPI_OK
No error
PAPI_EINVAL
Invalid argument
PAPI_ENOMEM
Insufficient memory
PAPI_ESYS
A system/C library call failed. Check errno variable
PAPI_ESBSTR
Substrate returned an error. E.g. unimplemented feature
PAPI_ECLOST
Access to the counters was lost or interrupted
PAPI_EBUG
Internal error
PAPI_ENOEVNT
Hardware event does not exist
PAPI_ECNFLCT
Hardware event exists, but resources are exhausted
PAPI_ENOTRUN
Event or envent set is currently counting
PAPI_EISRUN
Events or event set is currently running
PAPI_ENOEVST
No event set available
PAPI_ENOTPRESET
Argument is not a preset
PAPI_ENOCNTR
Hardware does not support counters
PAPI_EMISC
Any other error occured
52


PAPI Low-Level Interface

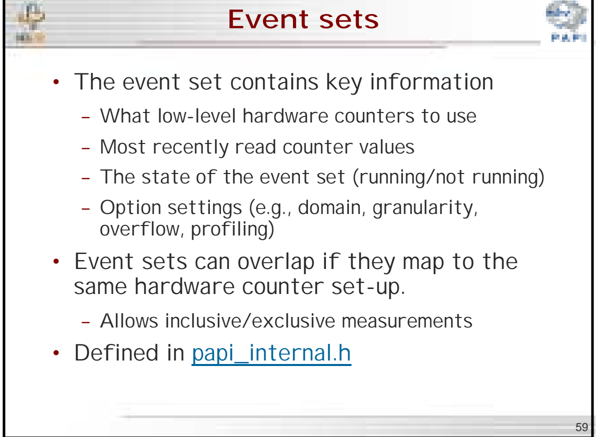

Low-level Interface
• Increased efficiency and functionality over the high level PAPI interface
• About 50 functions
(http://icl.cs.utk.edu/projects/papi/files/html_man/papi.html#4)
• Obtain information about the executable, the memory and the hardware
• Thread-safe
• Fully programmable (native events)
• Multiplexing
• Callbacks on counter overflow
• Profiling
54

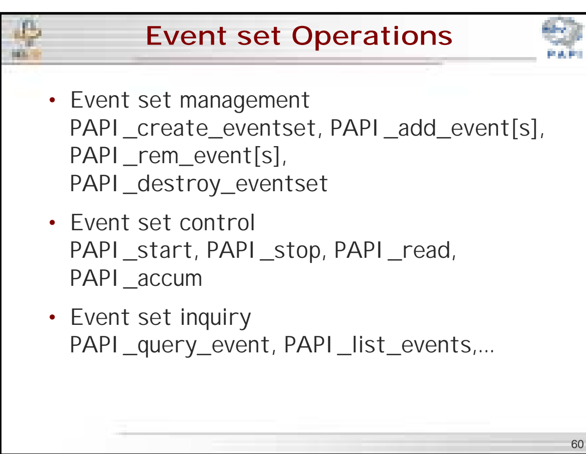


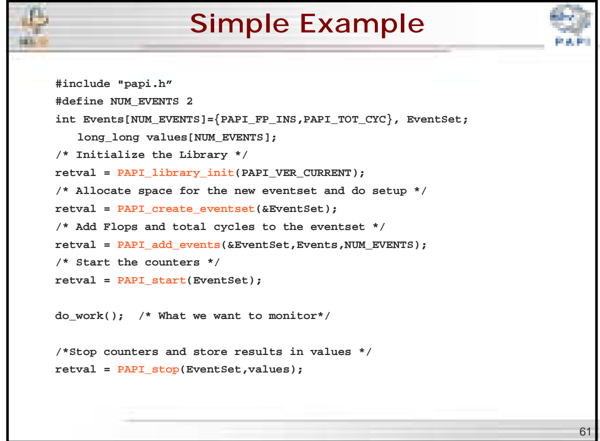


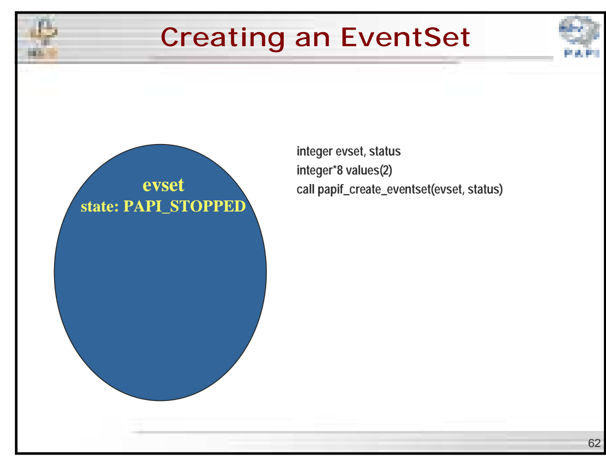


	Linux/ x86	Linux/ IA64	Cray T3E	IBM POWER3	MIPS R12K
PAPI start/stop (cycles/pair)	3524	22115	3325	14199	24850
PAPI read (cycles/call)	1299	6526	1514	3126	9810

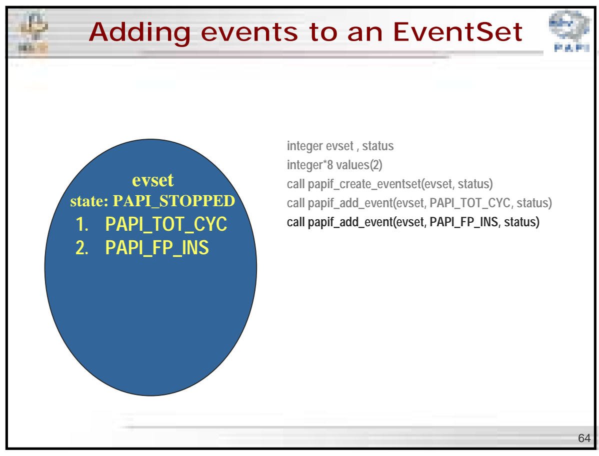


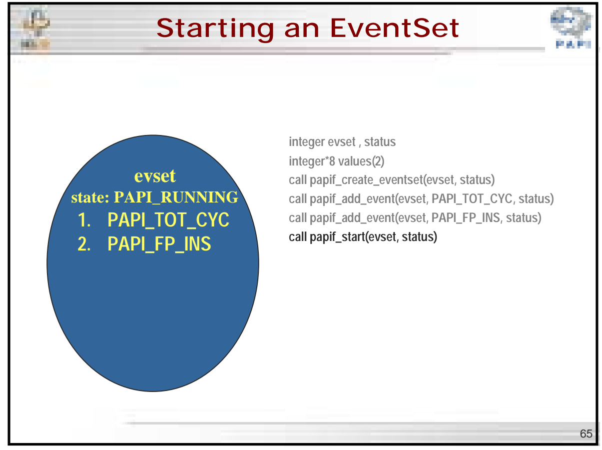


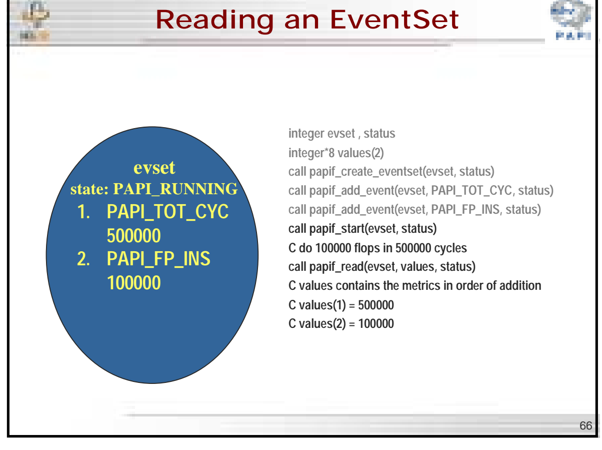

Simple Example
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC}, EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);
do_work(); /* What we want to monitor*/
/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
61

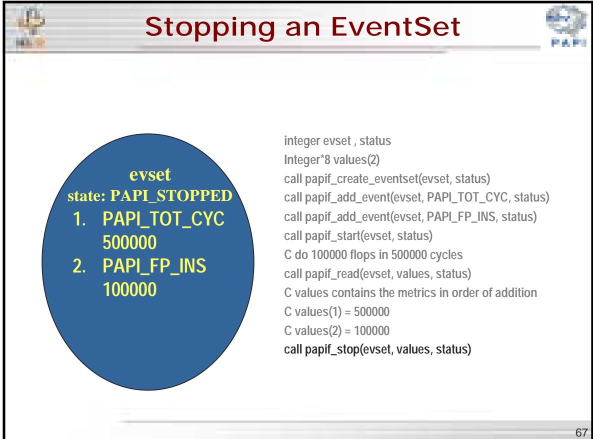

Creating an EventSet
evset
state: PAPI_STOPPED
integer evset, status
integer*8 values(2)
call papif_create_eventset(evset, status)
62

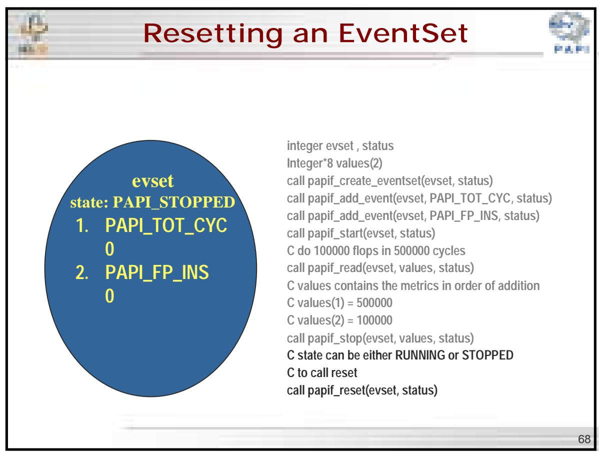
Adding events to an EventSet
evset
state: PAPI_STOPPED
1. PAPI_TOT_CYC
integer evset , status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
63

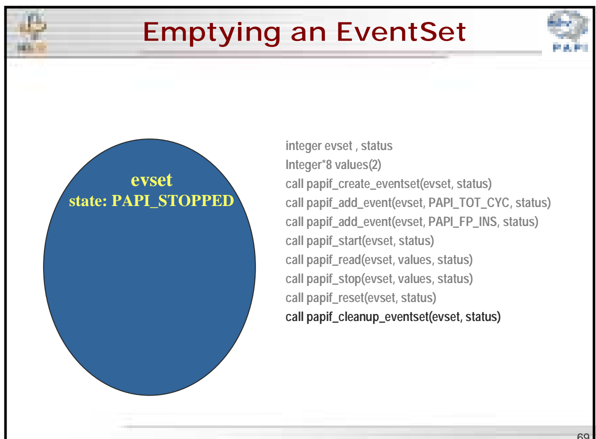

Adding events to an EventSet
evset
state: PAPI_STOPPED
1. PAPI_TOT_CYC
2. PAPI_FP_INS
integer evset , status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
64


Starting an EventSet
evset
state: PAPI_RUNNING
1. PAPI_TOT_CYC
2. PAPI_FP_INS
integer evset , status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
65

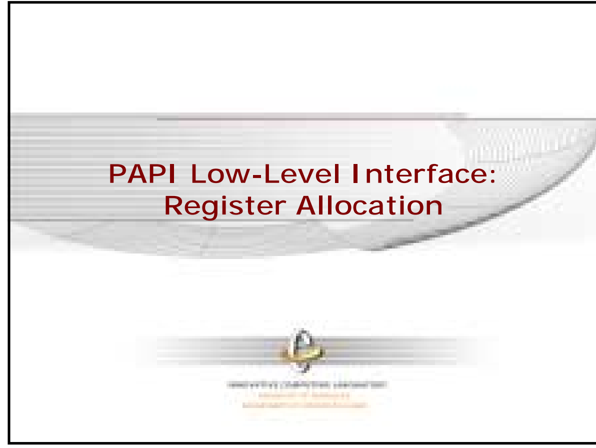

Reading an EventSet
evset
state: PAPI_RUNNING
1. PAPI_TOT_CYC
500000
2. PAPI_FP_INS
100000
integer evset , status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
C do 100000 flops in 500000 cycles
call papif_read(evset, values, status)
C values contains the metrics in order of addition
C values(1) = 500000
C values(2) = 100000
66

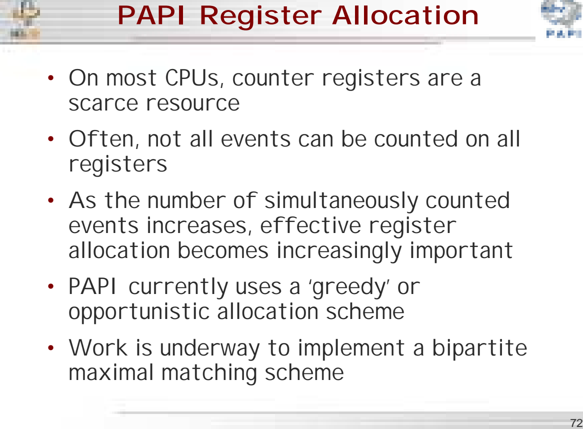

Stopping an EventSet
evset
state: PAPI_STOPPED
1. PAPI_TOT_CYC
500000
2. PAPI_FP_INS
100000
integer evset , status
Integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
C do 100000 flops in 500000 cycles
call papif_read(evset, values, status)
C values contains the metrics in order of addition
C values(1) = 500000
C values(2) = 100000
call papif_stop(evset, values, status)
67

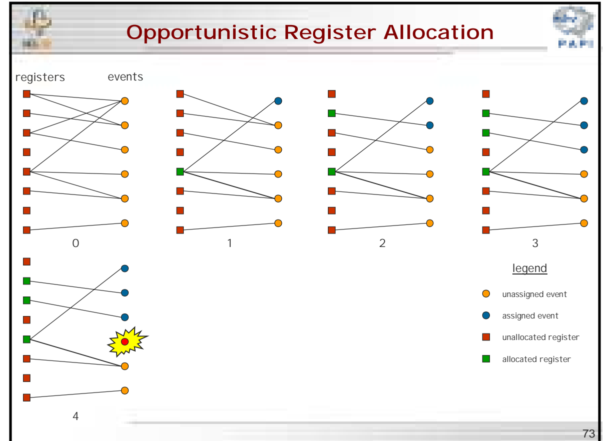

Resetting an EventSet
evset
state: PAPI_STOPPED
1. PAPI_TOT_CYC
0
2. PAPI_FP_INS
0
integer evset , status
Integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
C do 100000 flops in 500000 cycles
call papif_read(evset, values, status)
C values contains the metrics in order of addition
C values(1) = 500000
C values(2) = 100000
call papif_stop(evset, values, status)
C state can be either RUNNING or STOPPED
C to call reset
call papif_reset(evset, status)
68

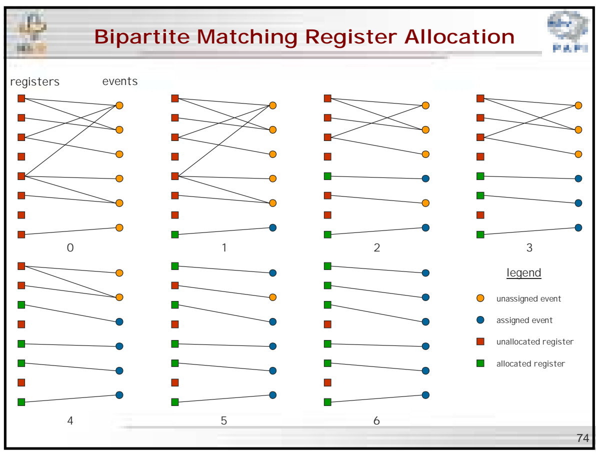

Emptying an EventSet
evset
state: PAPI_STOPPED
integer evset , status
Integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
call papif_stop(evset, values, status)
call papif_reset(evset, status)
call papif_cleanup_eventset(evset, status)
69

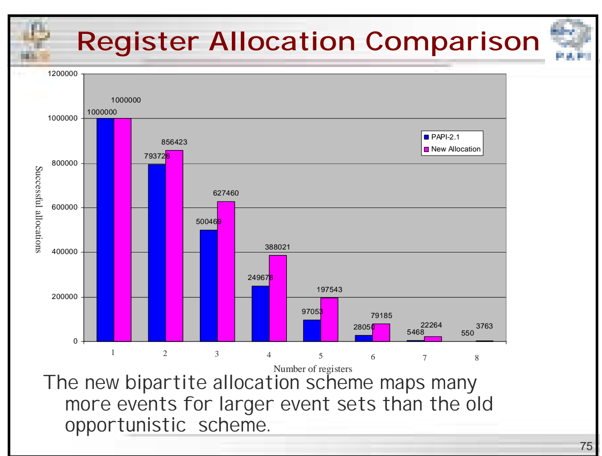
Freeing an EventSet
integer evset , status
Integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
call papif_stop(evset, values, status)
call papif_reset(evset, status)
call papif_cleanup_eventset(evset, status)
call papif_destroy_eventset(evset, status)
70

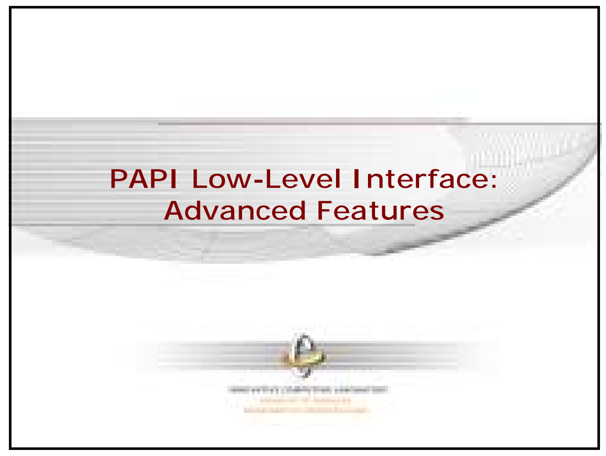

PAPI Low-Level Interface:
Register Allocation


PAPI Register Allocation
On most CPUs, counter registers are a scarce resource
Often, not all events can be counted on all registers
As the number of simultaneously counted events increases, effective register allocation becomes increasingly important
PAPI currently uses a 'greedy' or opportunistic allocation scheme
Work is underway to implement a bipartite maximal matching scheme
72









Using PAPI with Threads

- After PAPI_library_init, register a unique thread identifier function
- For Pthreads

```
retval=PAPI_thread_init(pthread_self, 0);
```

- OpenMP

```
retval=PAPI_thread_init(omp_get_thread_num, 0);
```

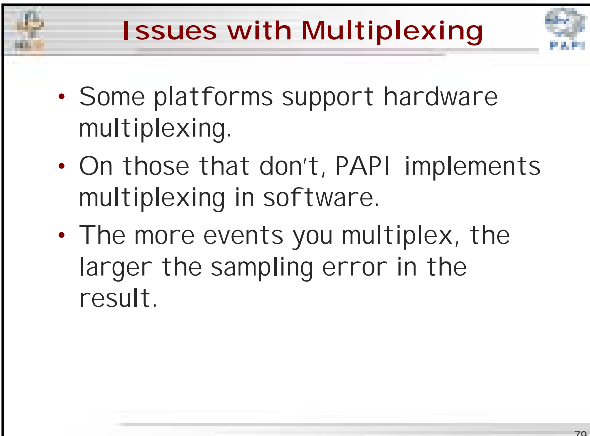
- Each thread is responsible for creation, start, stop and read of its own counters

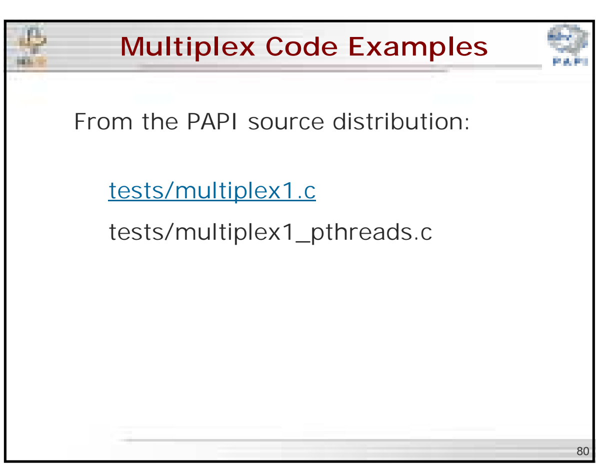
77

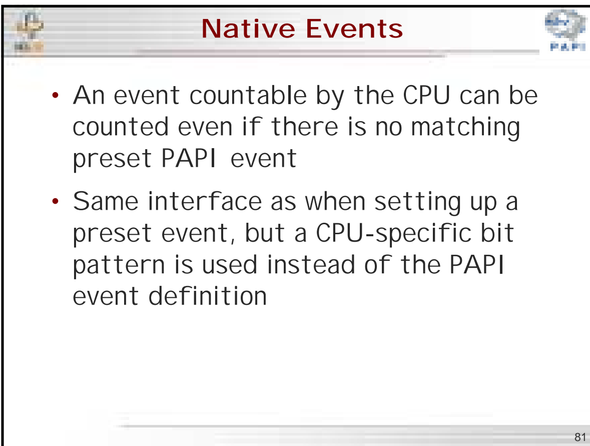
Using PAPI with Multiplexing

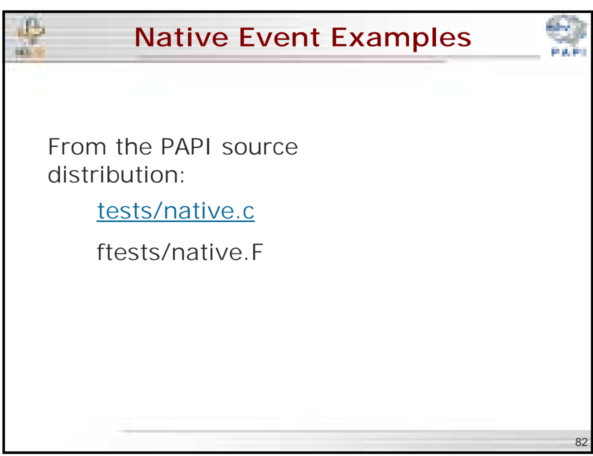
- Multiplexing allows simultaneous use of more counters than are supported by the hardware.
- PAPI_multiplex_init()
  - should be called after PAPI_library_init() to initialize multiplexing
- PAPI_set_multiplex(int *EventSet);
  - Used after the eventset is created to turn on multiplexing for that eventset
- Then use PAPI normally

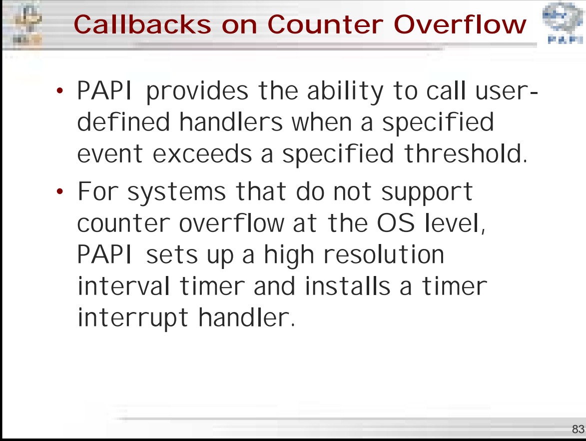
78

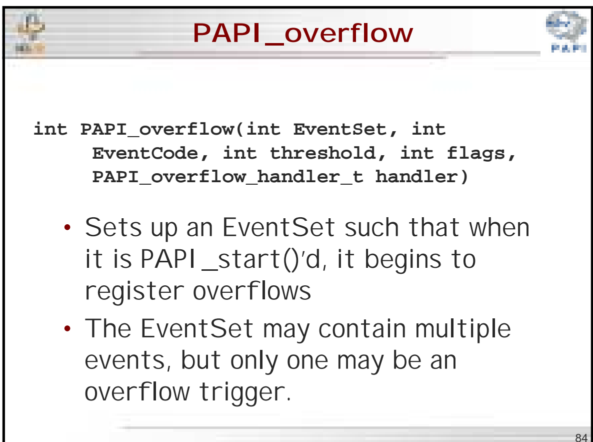

Issues with Multiplexing
• Some platforms support hardware multiplexing.
• On those that don't, PAPI implements multiplexing in software.
• The more events you multiplex, the larger the sampling error in the result.
79

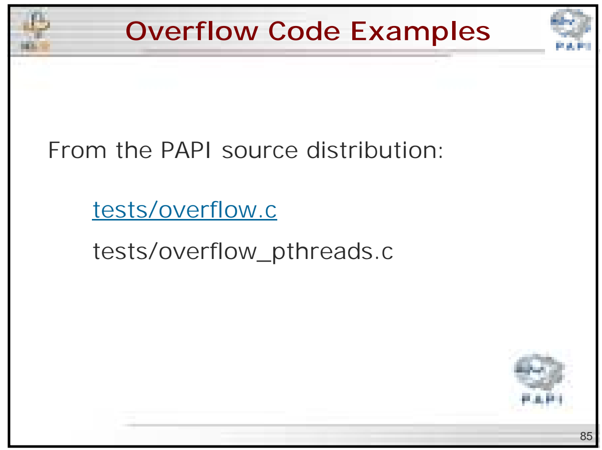

Multiplex Code Examples
From the PAPI source distribution:
tests/multiplex1.c
tests/multiplex1_pthreads.c
80

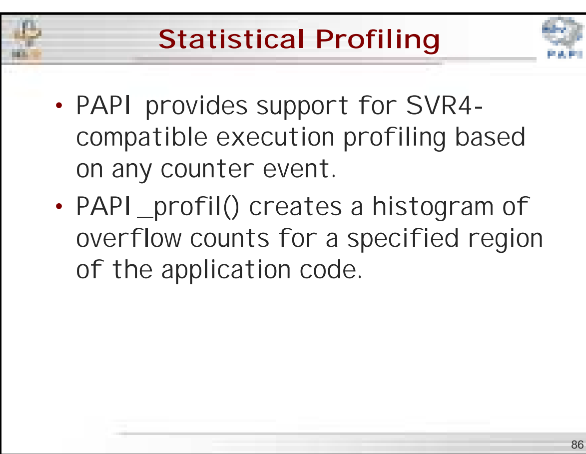

Native Events
• An event countable by the CPU can be counted even if there is no matching preset PAPI event
• Same interface as when setting up a preset event, but a CPU-specific bit pattern is used instead of the PAPI event definition
81

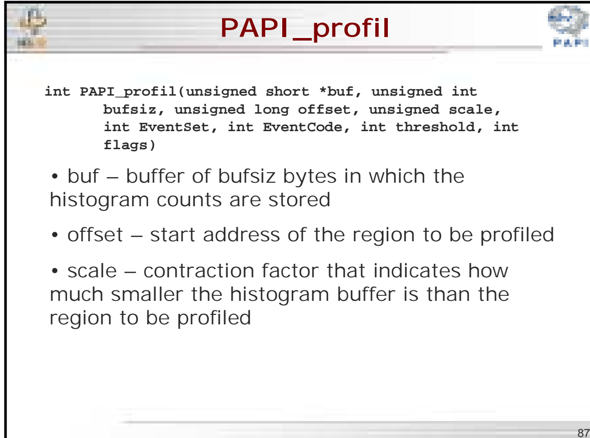

Native Event Examples
From the PAPI source distribution:
tests/native.c
ftests/native.F
82

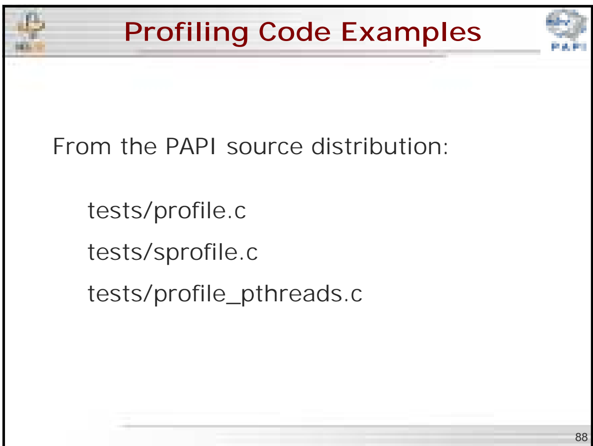

Callbacks on Counter Overflow
• PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.
• For systems that do not support counter overflow at the OS level, PAPI sets up a high resolution interval timer and installs a timer interrupt handler.
83


PAPI_overflow
int PAPI_overflow(int EventSet, int EventCode, int threshold, int flags, PAPI_overflow_handler_t handler)
• Sets up an EventSet such that when it is PAPI_start()'d, it begins to register overflows
• The EventSet may contain multiple events, but only one may be an overflow trigger.
84

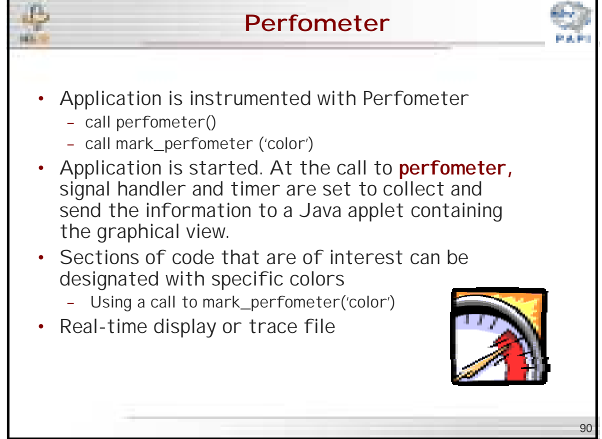

Overflow Code Examples
From the PAPI source distribution:
tests/overflow.c
tests/overflow_pthreads.c
85

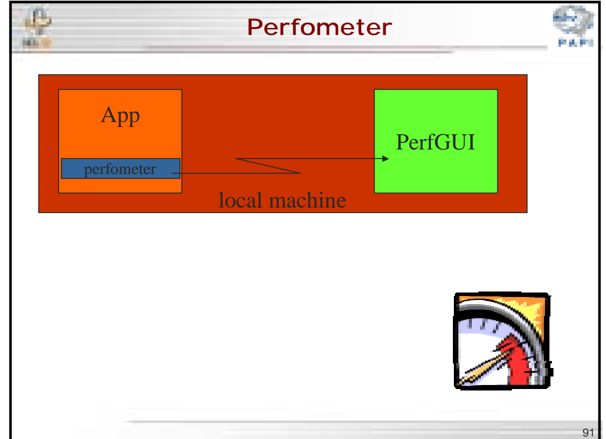

Statistical Profiling
• PAPI provides support for SVR4-compatible execution profiling based on any counter event.
• PAPI_profil() creates a histogram of overflow counts for a specified region of the application code.
86

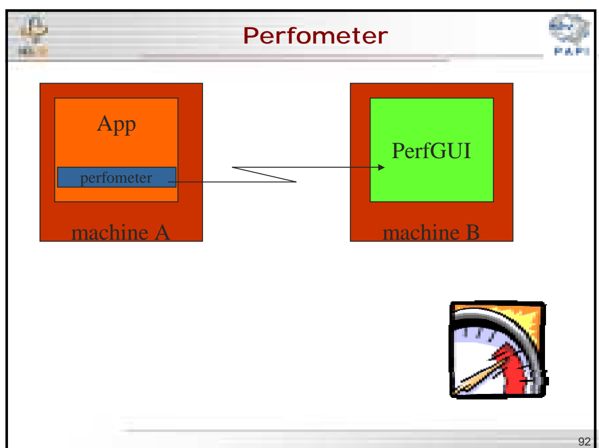

PAPI_profil
int PAPI_profil(unsigned short *buf, unsigned int bufsiz, unsigned long offset, unsigned scale, int EventSet, int EventCode, int threshold, int flags)
• buf – buffer of bufsiz bytes in which the histogram counts are stored
• offset – start address of the region to be profiled
• scale – contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled
87

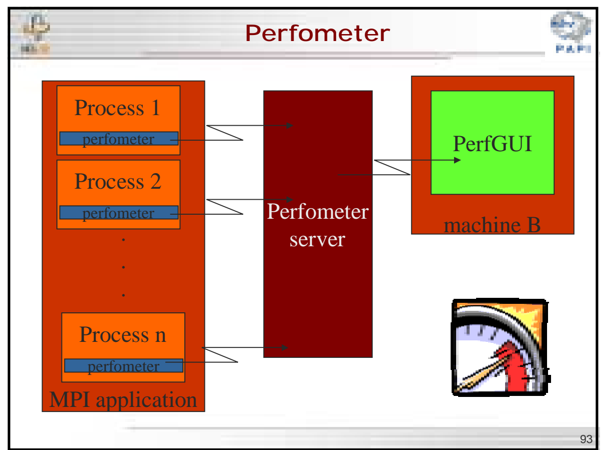

Profiling Code Examples
From the PAPI source distribution:
tests/profile.c
tests/sprofile.c
tests/profile_pthreads.c
88

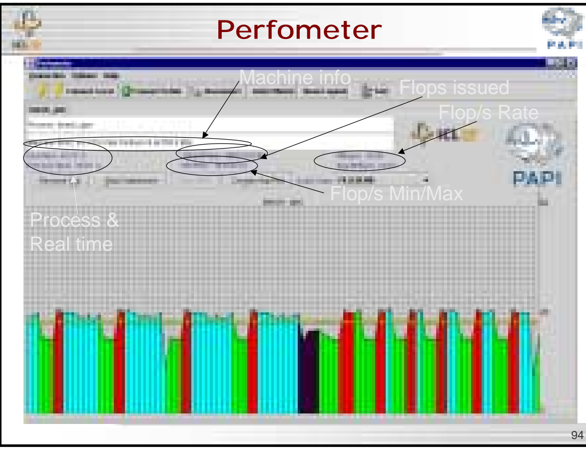
PAPI
INSIDE
Perfometer
A Qualitative Performance Visualization Tool

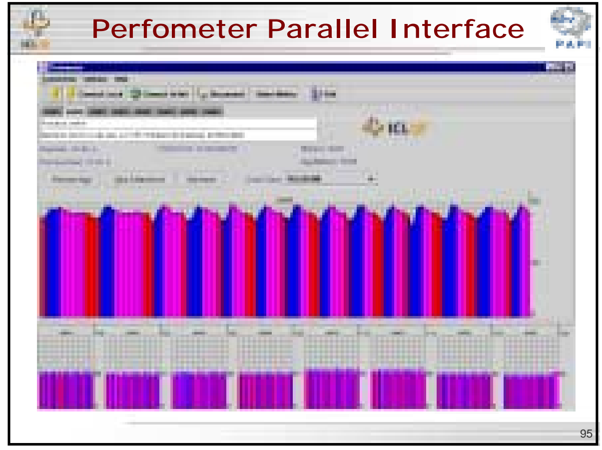

Perfometer
• Application is instrumented with Perfometer
– call perfometer()
– call mark_perfometer ('color')
• Application is started. At the call to perfometer, signal handler and timer are set to collect and send the information to a Java applet containing the graphical view.
• Sections of code that are of interest can be designated with specific colors
– Using a call to mark_perfometer('color')
• Real-time display or trace file
90

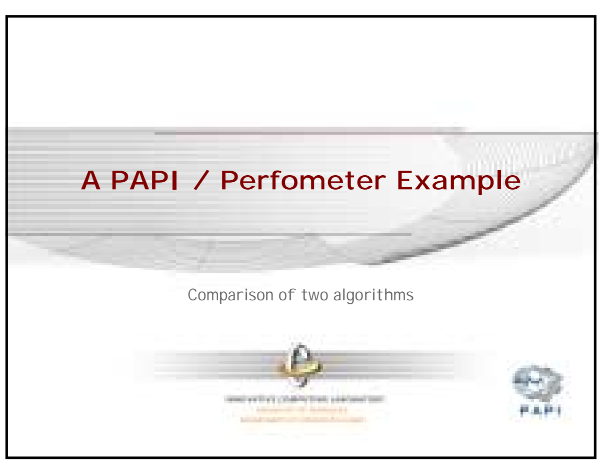

Perfometer
App
perfometer
PerfGUI
local machine
91

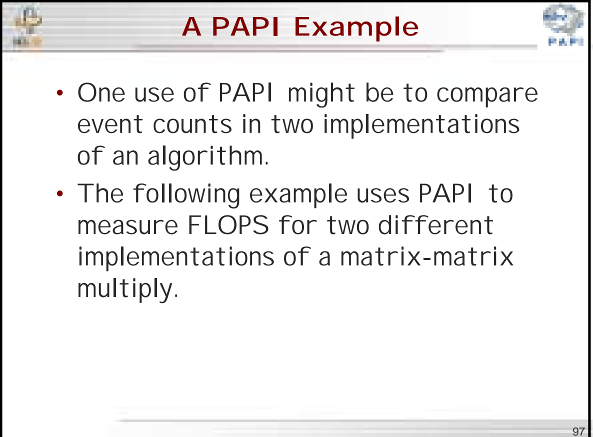

Perfometer
App
perfometer
machine A
PerfGUI
machine B
92

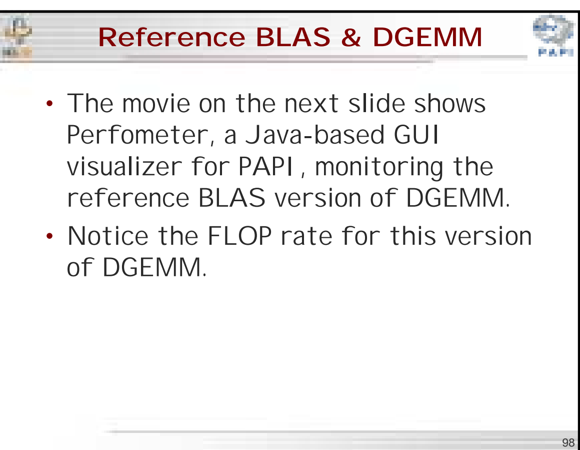

Perfometer
Process 1
perfometer
Process 2
perfometer
Process n
perfometer
MPI application
Perfometer
server
PerfGUI
machine B
93

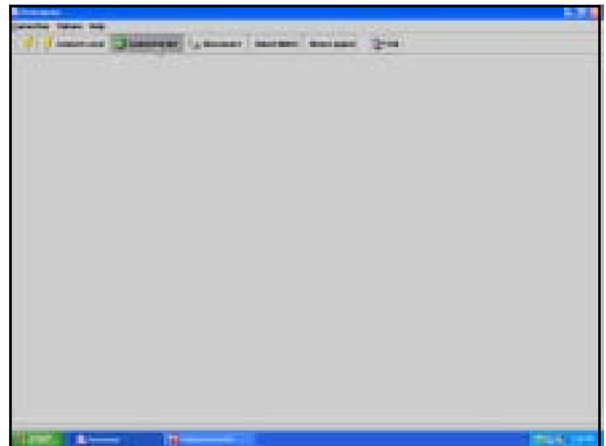

Perfometer
Machine info
Flops issued
Flop/s Rate
Flop/s Min/Max
Process &
Real time
PAPI
94

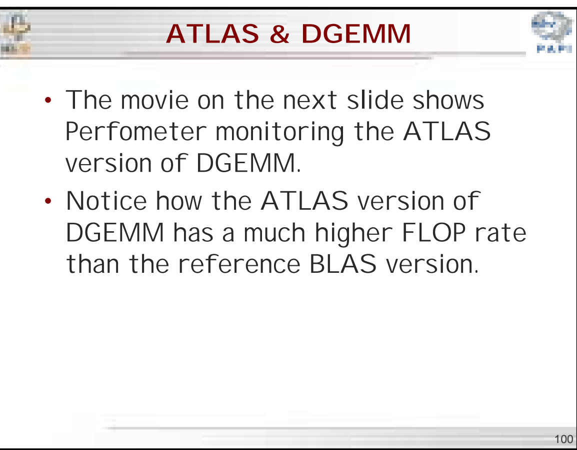

Perfometer Parallel Interface
95

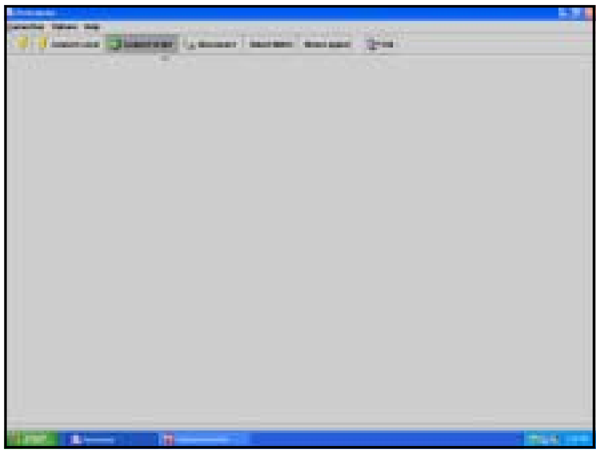

A PAPI / Perfometer Example
Comparison of two algorithms
PAPI

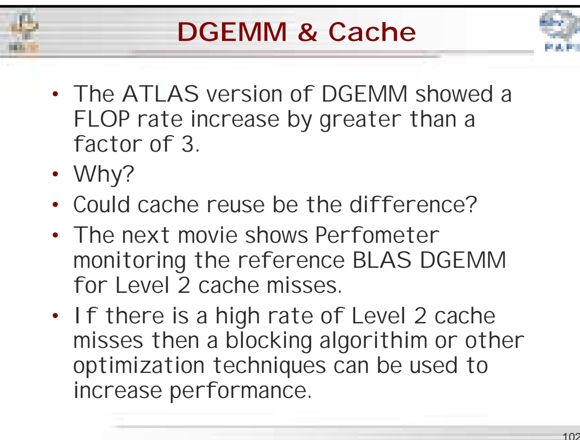

A PAPI Example
One use of PAPI might be to compare event counts in two implementations of an algorithm.
The following example uses PAPI to measure FLOPS for two different implementations of a matrix-matrix multiply.
97

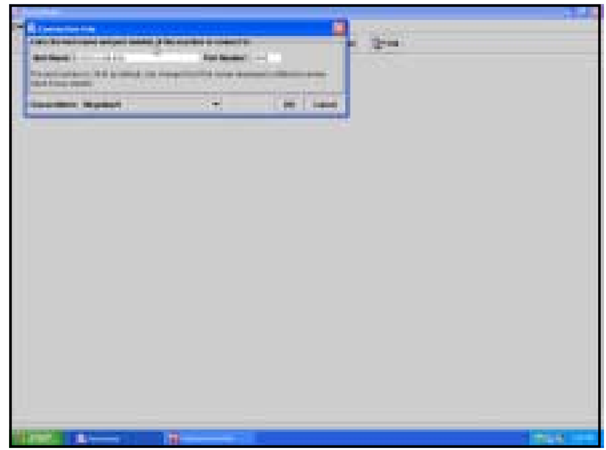

Reference BLAS & DGEMM
The movie on the next slide shows Perfometer, a Java-based GUI visualizer for PAPI, monitoring the reference BLAS version of DGEMM.
Notice the FLOP rate for this version of DGEMM.
98

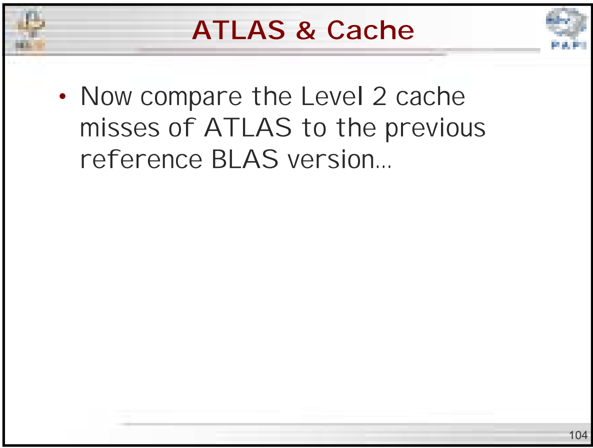


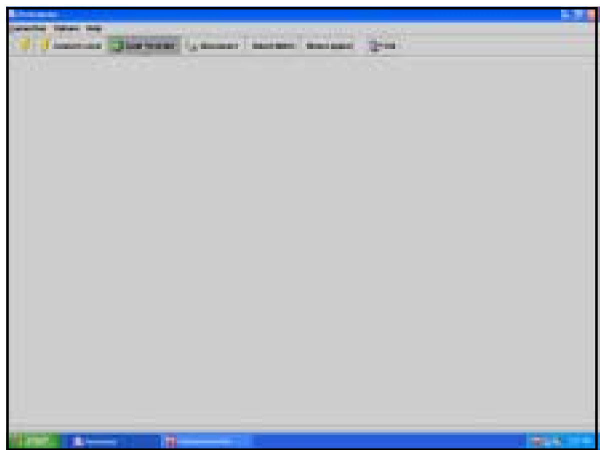

ATLAS & DGEMM
The movie on the next slide shows Perfometer monitoring the ATLAS version of DGEMM.
Notice how the ATLAS version of DGEMM has a much higher FLOP rate than the reference BLAS version.
100

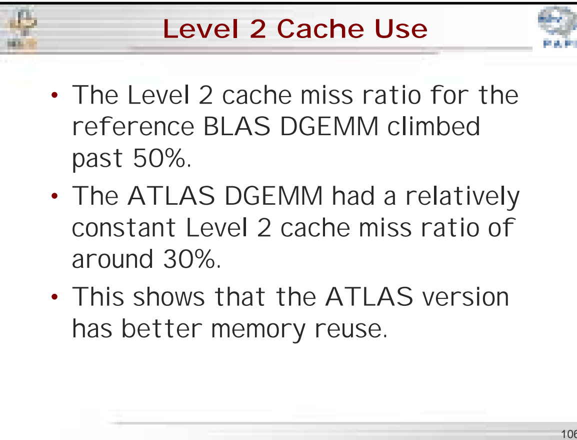


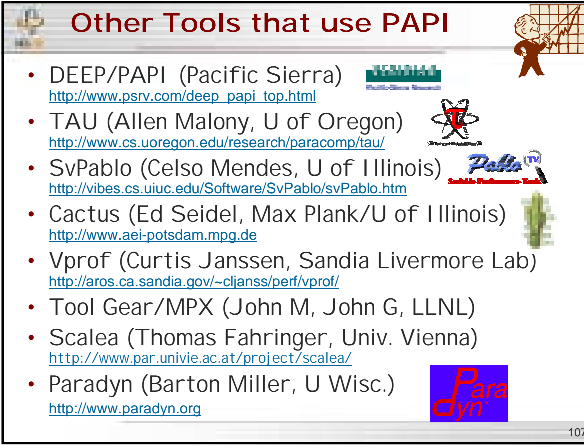

DGEMM & Cache
The ATLAS version of DGEMM showed a FLOP rate increase by greater than a factor of 3.
Why?
Could cache reuse be the difference?
The next movie shows Perfometer monitoring the reference BLAS DGEMM for Level 2 cache misses.
If there is a high rate of Level 2 cache misses then a blocking algorithm or other optimization techniques can be used to increase performance.
102

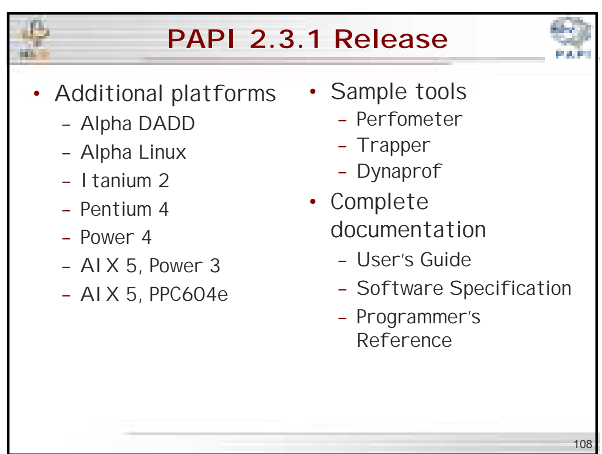


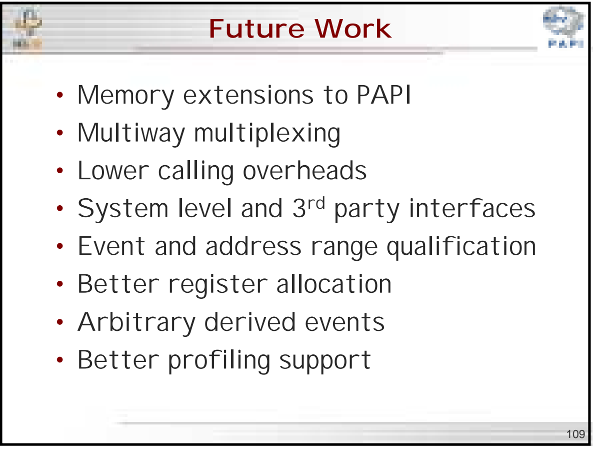

ATLAS & Cache
• Now compare the Level 2 cache misses of ATLAS to the previous reference BLAS version…
104

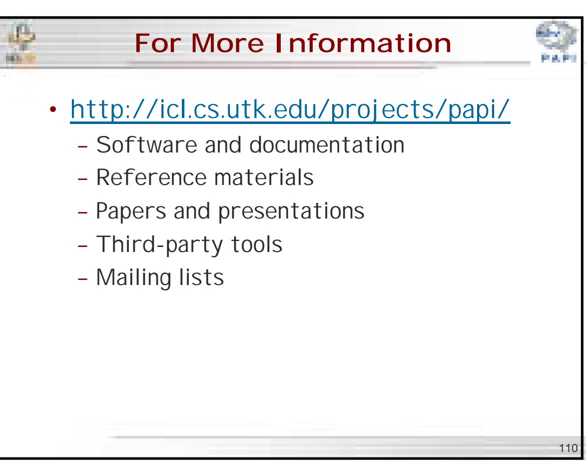


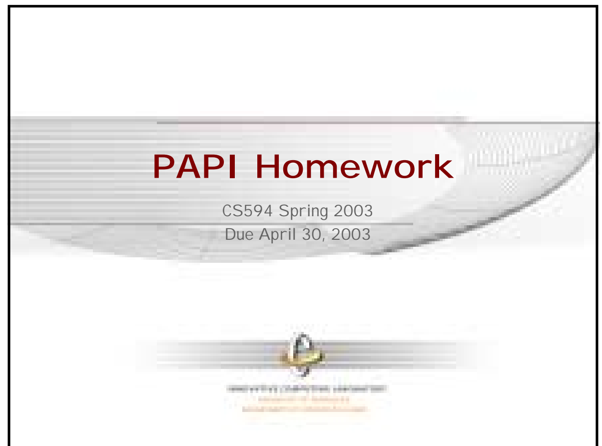

Level 2 Cache Use
• The Level 2 cache miss ratio for the reference BLAS DGEMM climbed past 50%.
• The ATLAS DGEMM had a relatively constant Level 2 cache miss ratio of around 30%.
• This shows that the ATLAS version has better memory reuse.
106

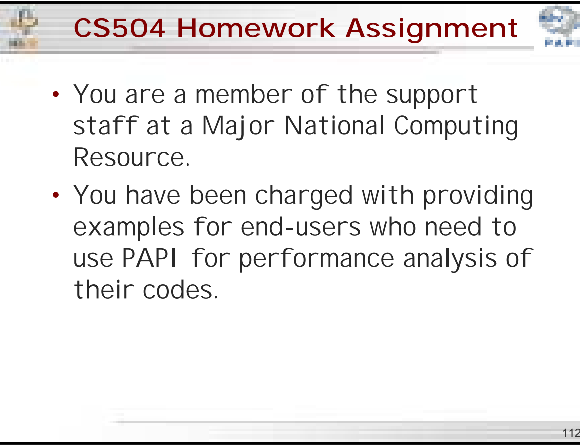

Other Tools that use PAPI
• DEEP/PAPI (Pacific Sierra)
http://www.psrv.com/deep_papi_top.html
• TAU (Allen Malony, U of Oregon)
http://www.cs.uoregon.edu/research/paracomp/tau/
• SvPablo (Celso Mendes, U of Illinois)
http://vibes.cs.uiuc.edu/Software/SvPablo/svPablo.htm
• Cactus (Ed Seidel, Max Plank/U of Illinois)
http://www.aei-potsdam.mpg.de
• Vprof (Curtis Janssen, Sandia Livermore Lab)
http://aros.ca.sandia.gov/~cljanss/perf/vprof/
• Tool Gear/MPX (John M, John G, LLNL)
• Scalea (Thomas Fahringer, Univ. Vienna)
http://www.par.univie.ac.at/project/scalea/
• Paradyn (Barton Miller, U Wisc.)
http://www.paradyn.org
107

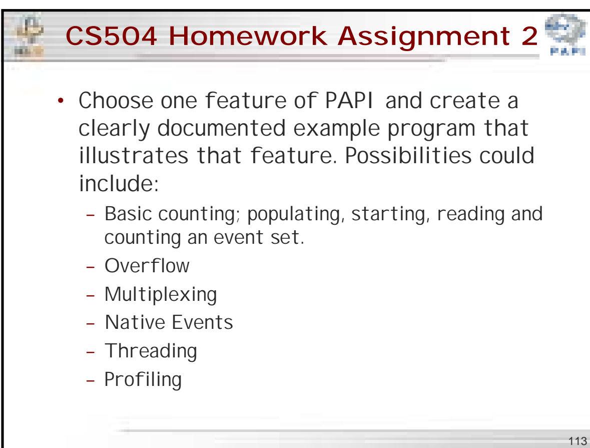

PAPI 2.3.1 Release
• Additional platforms
– Alpha DADD
– Alpha Linux
– Itanium 2
– Pentium 4
– Power 4
– AIX 5, Power 3
– AIX 5, PPC604e
• Sample tools
– Perfometer
– Trapper
– Dynaprof
• Complete documentation
– User's Guide
– Software Specification
– Programmer's Reference
108

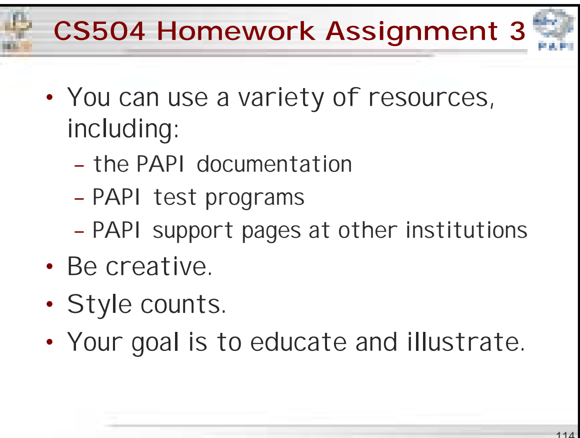

Future Work
• Memory extensions to PAPI
• Multiway multiplexing
• Lower calling overheads
• System level and 3rd party interfaces
• Event and address range qualification
• Better register allocation
• Arbitrary derived events
• Better profiling support
109


For More Information
• http://icl.cs.utk.edu/projects/papi/
– Software and documentation
– Reference materials
– Papers and presentations
– Third-party tools
– Mailing lists
110


PAPI Homework
CS594 Spring 2003
Due April 30, 2003


CS504 Homework Assignment
• You are a member of the support staff at a Major National Computing Resource.
• You have been charged with providing examples for end-users who need to use PAPI for performance analysis of their codes.
112


CS504 Homework Assignment 2
• Choose one feature of PAPI and create a clearly documented example program that illustrates that feature. Possibilities could include:
– Basic counting; populating, starting, reading and counting an event set.
– Overflow
– Multiplexing
– Native Events
– Threading
– Profiling
113


CS504 Homework Assignment 3
• You can use a variety of resources, including:
– the PAPI documentation
– PAPI test programs
– PAPI support pages at other institutions
• Be creative.
• Style counts.
• Your goal is to educate and illustrate.
114

CS504 Homework Assignment 4

- Test your example program on any ICL machine with PAPI installed:
  - TORC1-8; MSC01-08; any others
- Submit via email the following:
  - Well documented(!) source code
  - Outputs from the example program
  - Supporting descriptions, discussion, or documentation of the example as needed
- Email to: terpstra@cs.utk.edu by April 30, 2003

115