

# CS594: Performance Evaluation and Tuning

Vince Weaver

`vweaver1@eecs.utk.edu`

15 February 2012

Some slides based on previous years presentations by Heike Jagode

# Introduction to Performance Analysis



# What is Performance?

- Getting results as quickly as possible?
- Getting *correct* results as quickly as possible?
- What about Budget?
- What about Development Time?
- What about Hardware Usage?
- What about Power Consumption?

# Motivation

## HPC environments are expensive:

- Procurement costs: ~\$40 million
- Operational costs: ~\$5 million/year
- Electricity costs: 1 MW / year ~\$1 million
- Air Conditioning costs: ??

# PART II — Computer Architecture Review

You don't have to be a computer architect to optimize code. . .

But it helps.



**In an ideal world. . .**

**Code  
+  
Input**



**Mystery  
Box**

**Results**



# Compiler Should Not be Neglected

```
int init_array(double c) {  
    int i,j;  
  
    for(i=0;i<100;i++) {  
        for(j=0;j<100;j++) {  
            A[i][j]=(double)(i*j)+c;  
        }  
    }  
}
```

# gcc -S

## no optimization

```
init_array:
    pushq    %rbp
    movq     %rsp, %rbp
    movsd    %xmm0, -24(%rbp)
    movl     $0, -4(%rbp)
    jmp      .L2
.L5:        movl     $0, -8(%rbp)
    jmp      .L3
.L4:        movl     -4(%rbp), %edx
    movl     -8(%rbp), %ecx
    movl     -4(%rbp), %eax
    imull    -8(%rbp), %eax
    cvtsi2sd          %eax, %xmm0
    addsd    -24(%rbp), %xmm0
    movslq   %ecx,%rcx
    movslq   %edx,%rdx
    movq     %rdx, %rax
    salq     $2, %rax
    ...
```

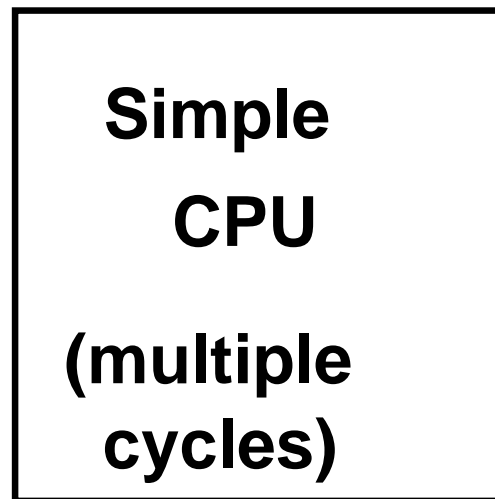
## -O3 -march=native

```
init_array:
    movl     $A, %eax
    xorl     %ecx, %ecx
    movdqa   .LC0(%rip), %xmm5
    movddup  %xmm0, %xmm3
    movdqa   .LC1(%rip), %xmm6
.L2:        movd     %ecx, %xmm0
    movdqa   %xmm6, %xmm2
    pshufd   $0, %xmm0, %xmm4
    leaq     800(%rax), %rdx
.L3:        movdqa   %xmm2, %xmm0
    padd     %xmm5, %xmm2
    pmulld   %xmm4, %xmm0
    cvtdq2pd          %xmm0, %xmm1
    pshufd   $238, %xmm0, %xmm0
    addpd    %xmm3, %xmm1
    cvtdq2pd          %xmm0, %xmm0
    movapd   %xmm1, (%rax)
    ...
```



# Simple CPU

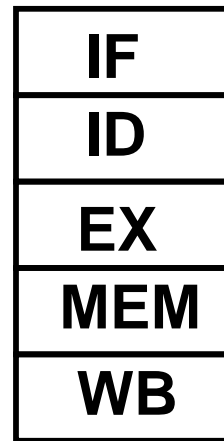
**1 instruction at a time**



Optimization primarily cycle-counting.

# Pipelined CPU

5 instructions "in Flight"



Optimization: avoid Load and Branch Delays, "Bubbles" in pipeline.

# Branch Prediction

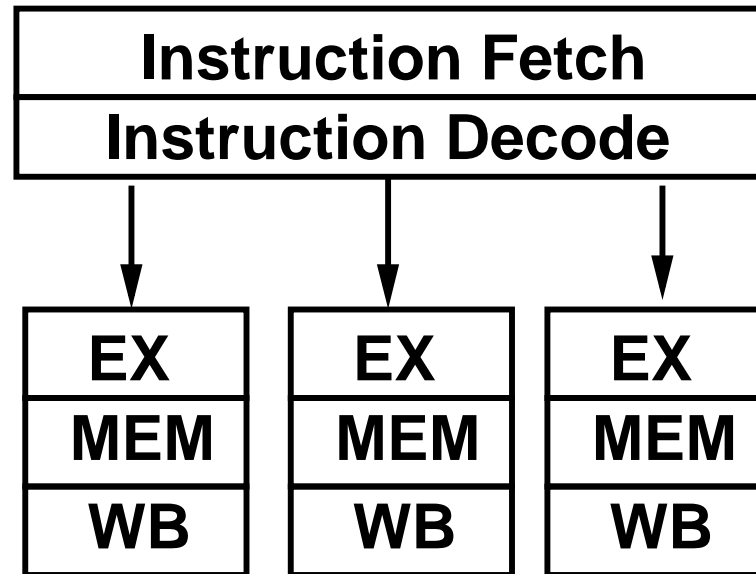
- As pipelines get deeper, more work lost if branch mispredicted
- Modern branch predictors very good (  $>95\%$  accuracy) on regular workloads
- Various implementations; from simple static (backward taken) to 2-bit saturating counters, to multi-level “tournament” predictors
- Optimizations: Loop Unrolling

# Loop Unrolling

```
for (i=0; i<10; i++) {  
    A[i]=i*i;  
}
```

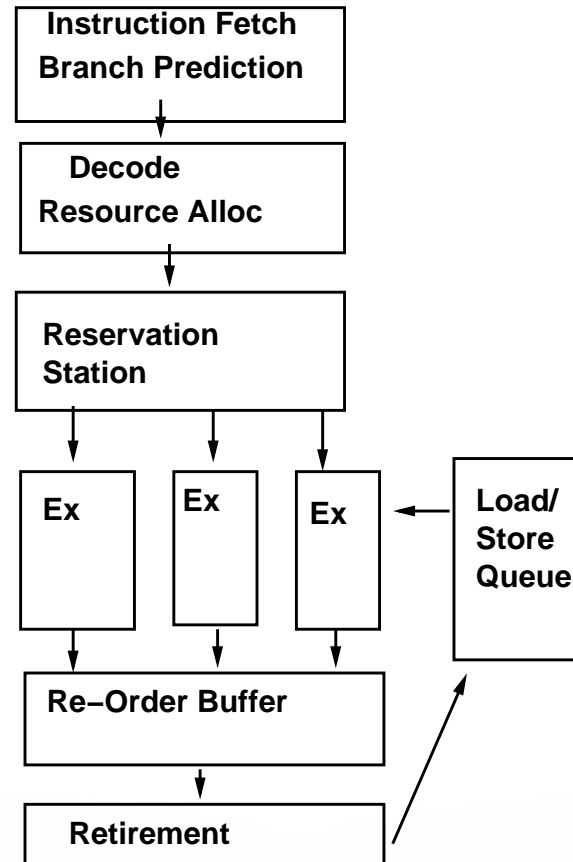
```
A[0]=0;  
A[1]=1;  
A[2]=4;  
A[3]=9;  
A[4]=16;  
A[5]=25;  
A[6]=36;  
A[7]=49;  
A[8]=64;  
A[9]=81;
```

# Super-Scalar CPU



Optimization: pair instructions that can run together,  
avoid data hazards

# Out of Order



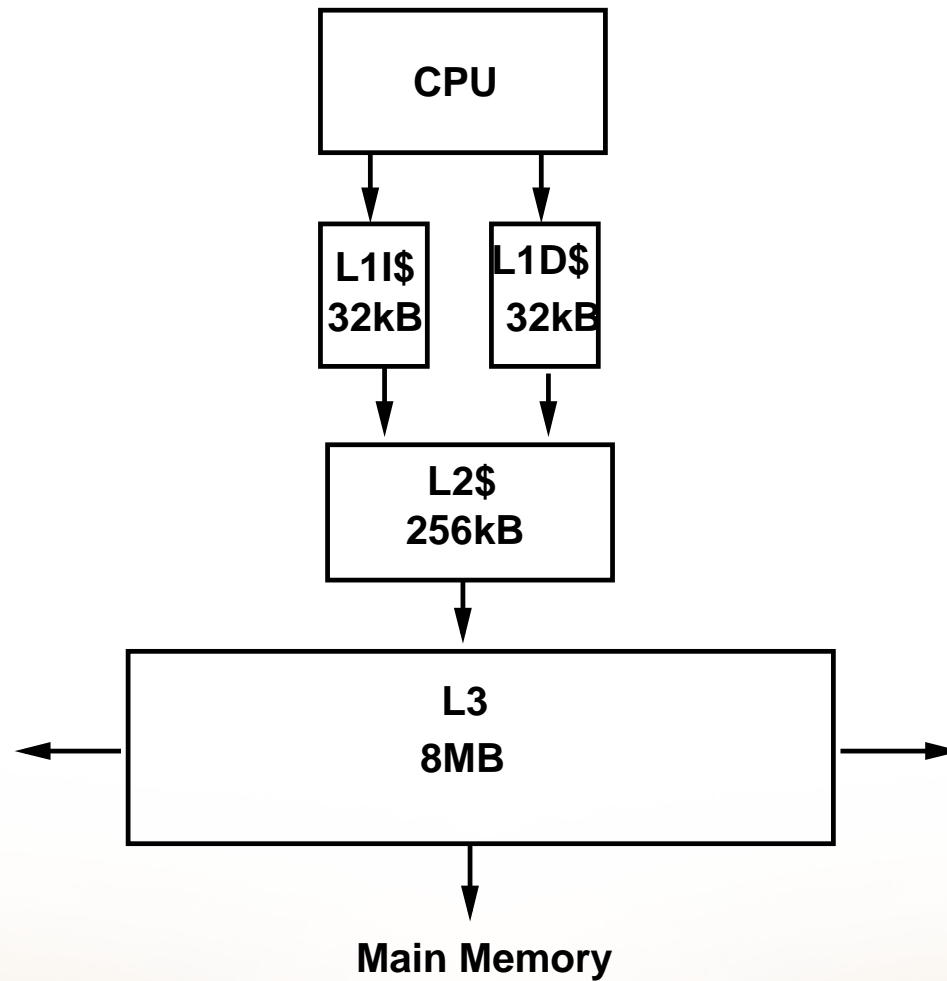
Tries to solve performance issues in hardware.

# Memory Hierarchy

There's never enough memory, so a hierarchy is created of increasingly slow storage.


- Older: CPU → Memory → Disk → Tape
- Old: CPU → L1 Cache → Memory → Disk
- Now?: CPU → L1/L2/L3 Cache → Memory → SSD  
Disk → Network/Cloud

# Caches






# Cache Example

Line	Tag	Data			
0x00					
0x01					
0x02					
0x03					
					
0xff					

# Cache Example

Access: 

0x00120	01	2
---------	----	---

Line	Tag	Data			
0x00					
0x01					
0x02					
0x03					
					
0xff					

# Cache Example

Access: 

0x00120	01	2
---------	----	---

**MISS**

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02					
0x03					
⋮					
0xff					

# Cache Example

Access: 

0x99abc	02	03
---------	----	----

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02					
0x03					
⋮					
0xff					

# Cache Example

Access: 

0x99abc	02	03
---------	----	----

**MISS**

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x99abc	Z	Q	R	T
0x03					
⋮					
0xff					

# Cache Example

Access:	0x00120	01	4
---------	---------	----	---

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x99abc	Z	Q	R	T
0x03					
⋮					
0xff					

# Cache Example

Access: 

0x00120	01	4
---------	----	---

HIT!

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x99abc	Z	Q	R	T
0x03					
⋮					
0xff					

# Cache Example

Access: 

0x00120	02	4
---------	----	---

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x99abc	Z	Q	R	T
0x03					
⋮					
0xff					



# Cache Example

Access: 

0x00120	02	4
---------	----	---

**MISS!**

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x00120	E	F	G	H
0x03					
⋮					
0xff					

# Cache Example

Access:	0x99abc	02	1
---------	---------	----	---

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x00120	E	F	G	H
0x03					
⋮					
0xff					

# Cache Example

Access: 

0x99abc	02	1
---------	----	---

**MISS**

Line	Tag	Data			
0x00					
0x01	0x00120	A	B	C	D
0x02	0x99abc	Z	Q	R	T
0x03					
⋮					
0xff					

# Cache Miss Types

- Compulsory — can't avoid, first time seen
- Capacity — wouldn't have been a miss with larger cache
- Conflict — miss caused by conflict with another address
- Coherence — miss caused by other processor

# Fixing Compulsory Misses

- Prefetching (Hardware and Software)

# Fixing Capacity Misses

- Build Bigger Caches

# Fixing Conflict Misses

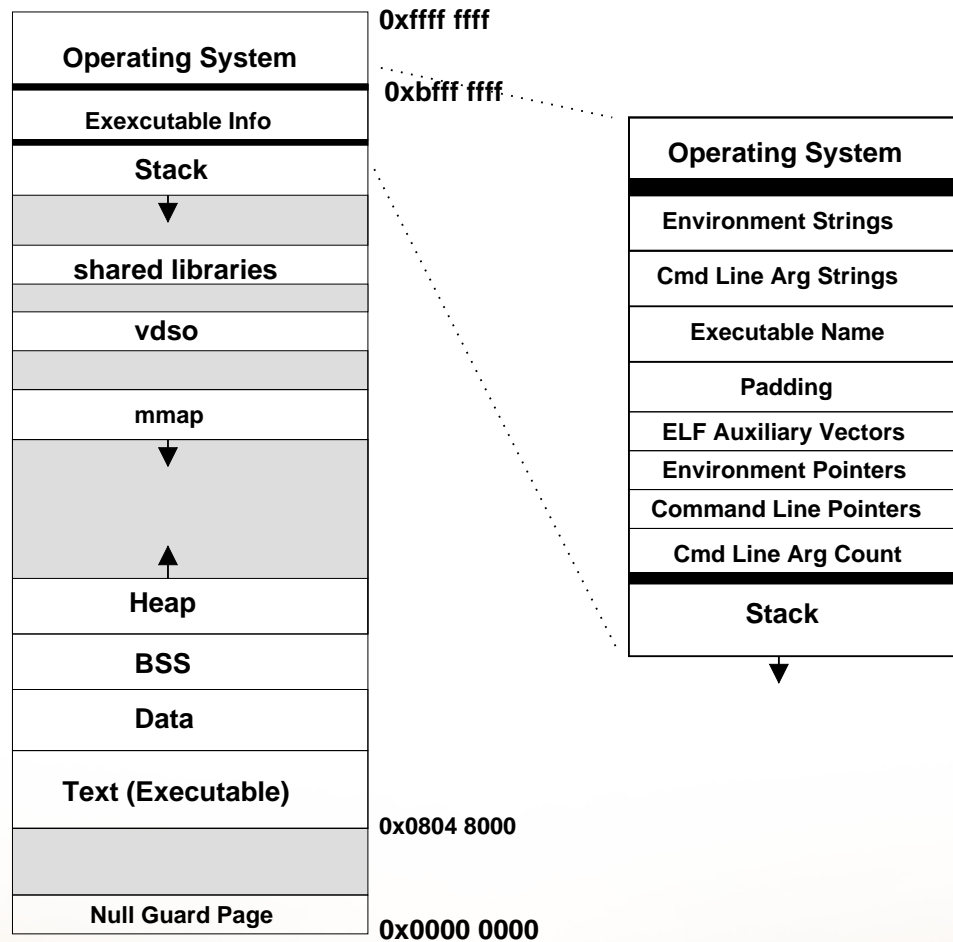
- More Ways in Cache
- Code/Variable Alignment

# Fixing Coherence Misses

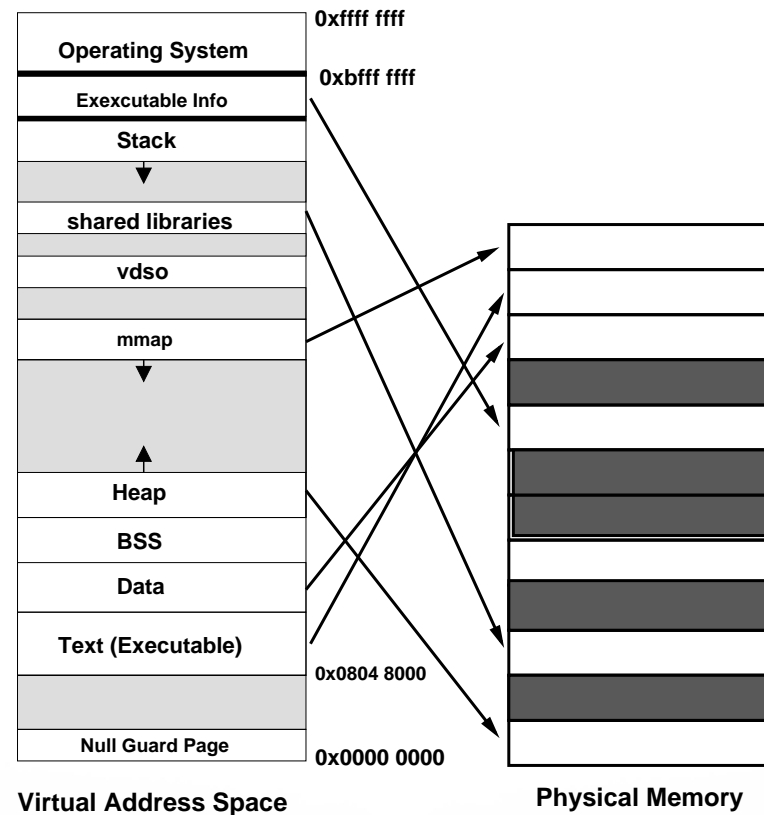
- False Sharing



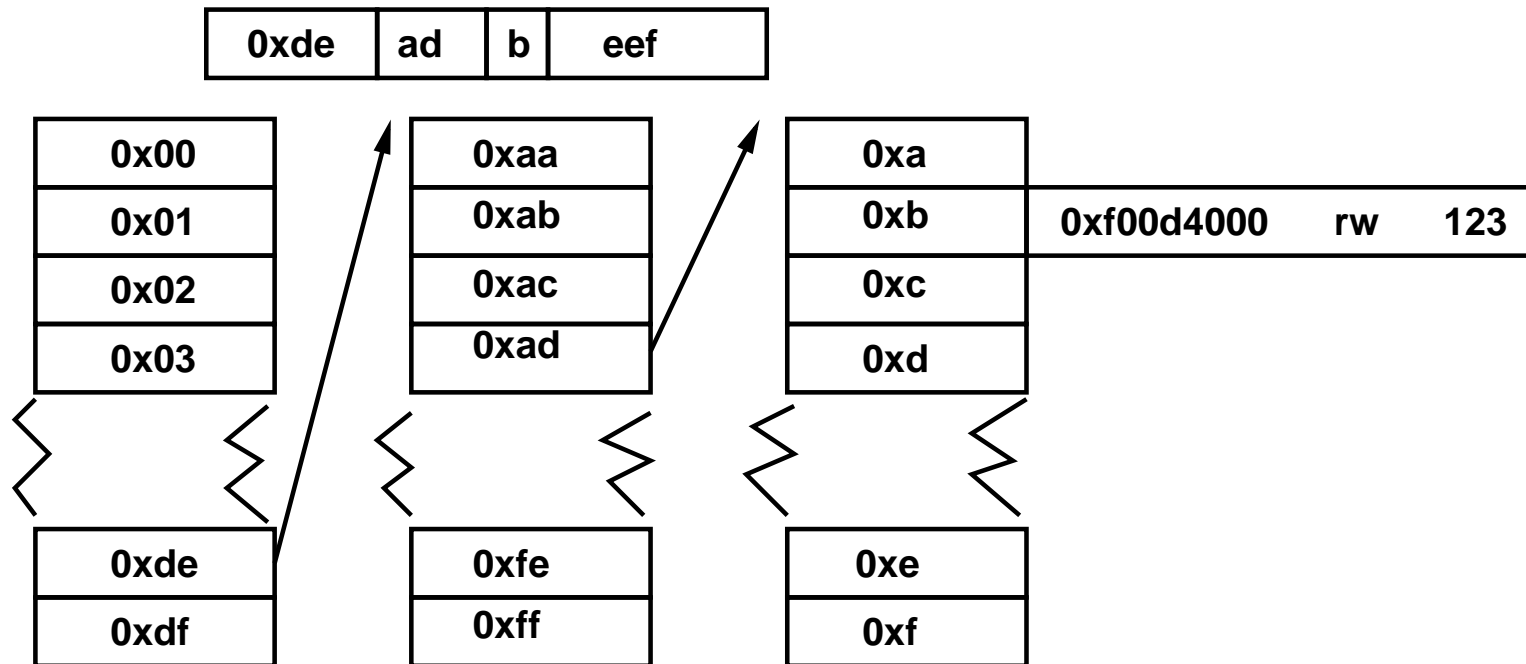
# Virtual Memory



# Virtual/Physical Mapping



# Page Tables



# TLB

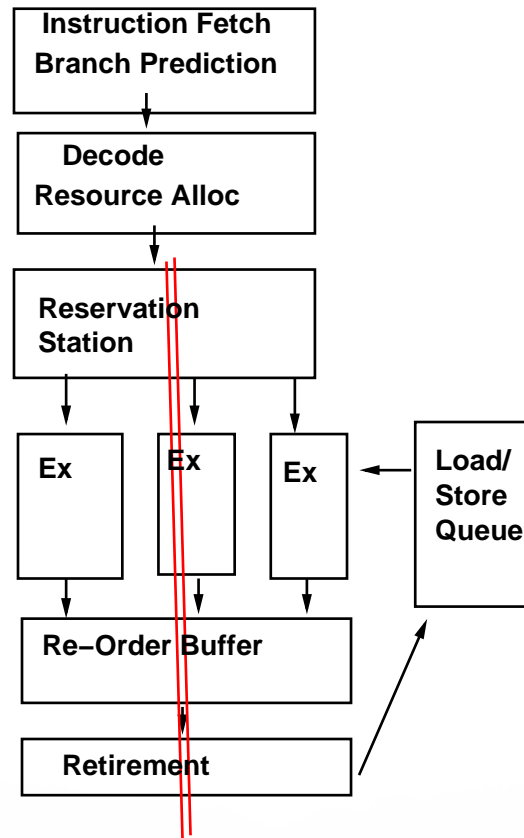
0xde	ad	b	eef
------	----	---	-----

Virt Addr	Phys Addr	perm	v	pid
0x12345000	0x1ee70000	x	y	456
0xdeadb000	0xf00d4000	rw	y	123
0xffff0000	0x00000000	-	n	0
0xfeedf0000	0xcafef0000	r	y	456

# Vector Units

- Operate on many values in parallel
- MMX/SSE/SSE2/SSE3/SSSE3/SSE4
- AVX
- AltiVec, NEON

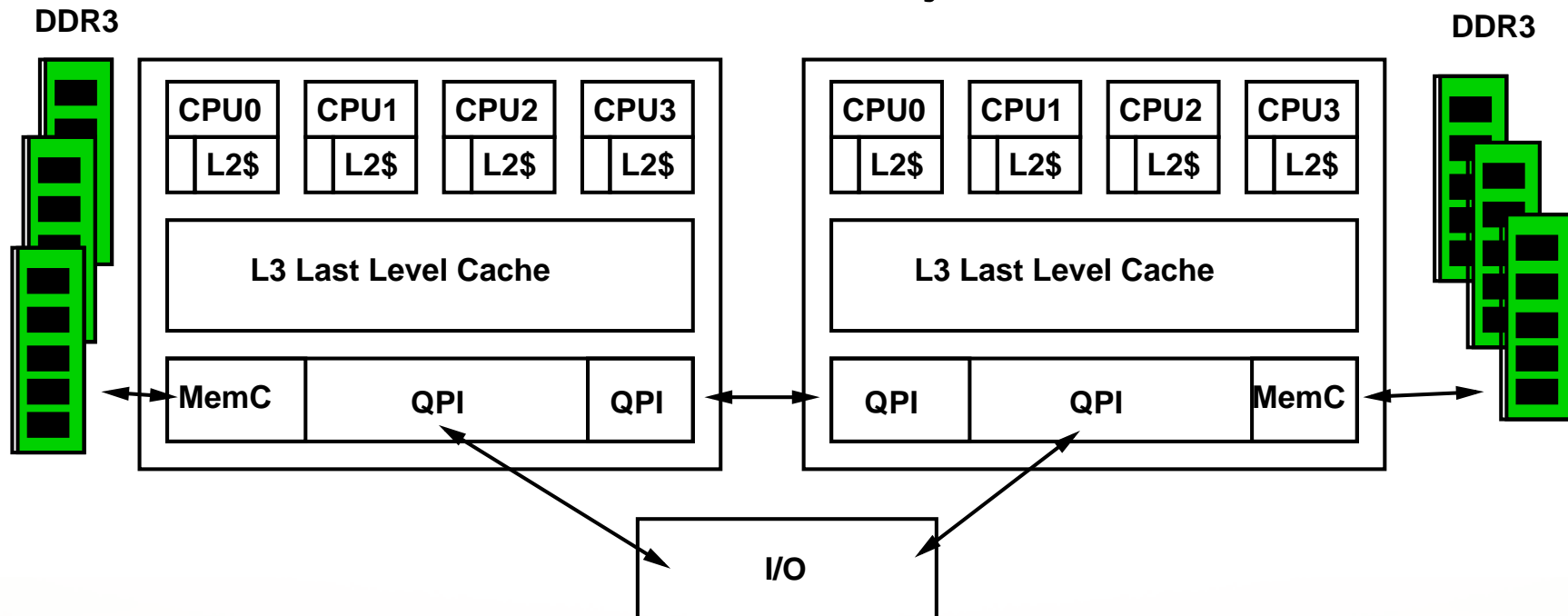
# Multi-Threading



Not using all hardware? Split into two.

# Multi-core

Nehalem CPU, as in Hydra machines



# Memory Latency Concerns

- Cache Coherency — MESI
- NUMA — Non-Uniform Memory Access



## More Info

- Performance Analysis Guide for Intel Core i7 Processors and Intel Xeon 5500 Processors by David Levinthal
- Intel 64 and IA-32 Architectures Optimization Reference Manual
- Software Optimization Guide for AMD Family 15h Processors

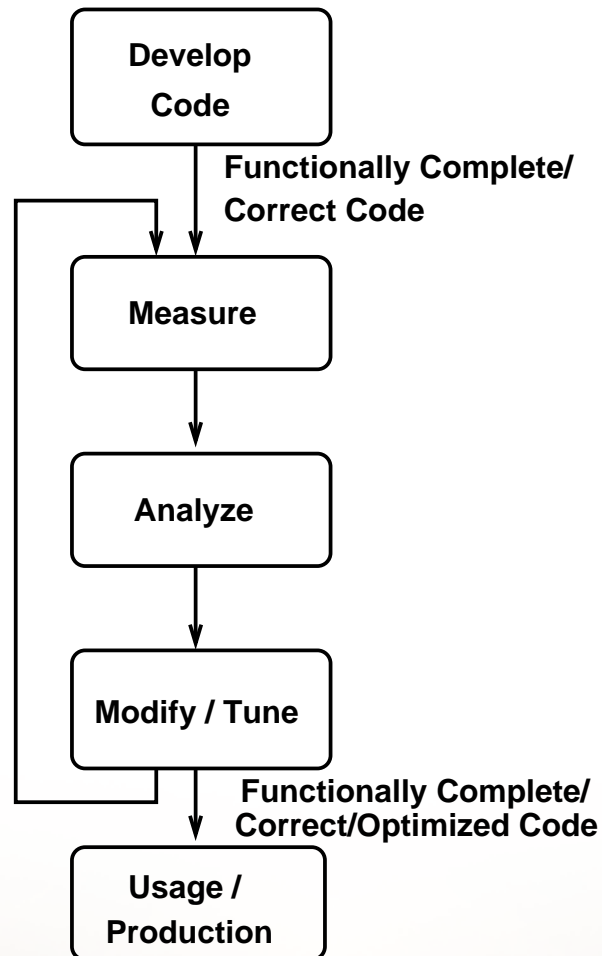
# PART III — Types of Performance Analysis



# Know Your Limitation

- CPU Constrained
- Memory Constrained (Memory Wall)
- I/O Constrained
- Thermal Constrained
- Energy Constrained

# Performance Optimization Cycle



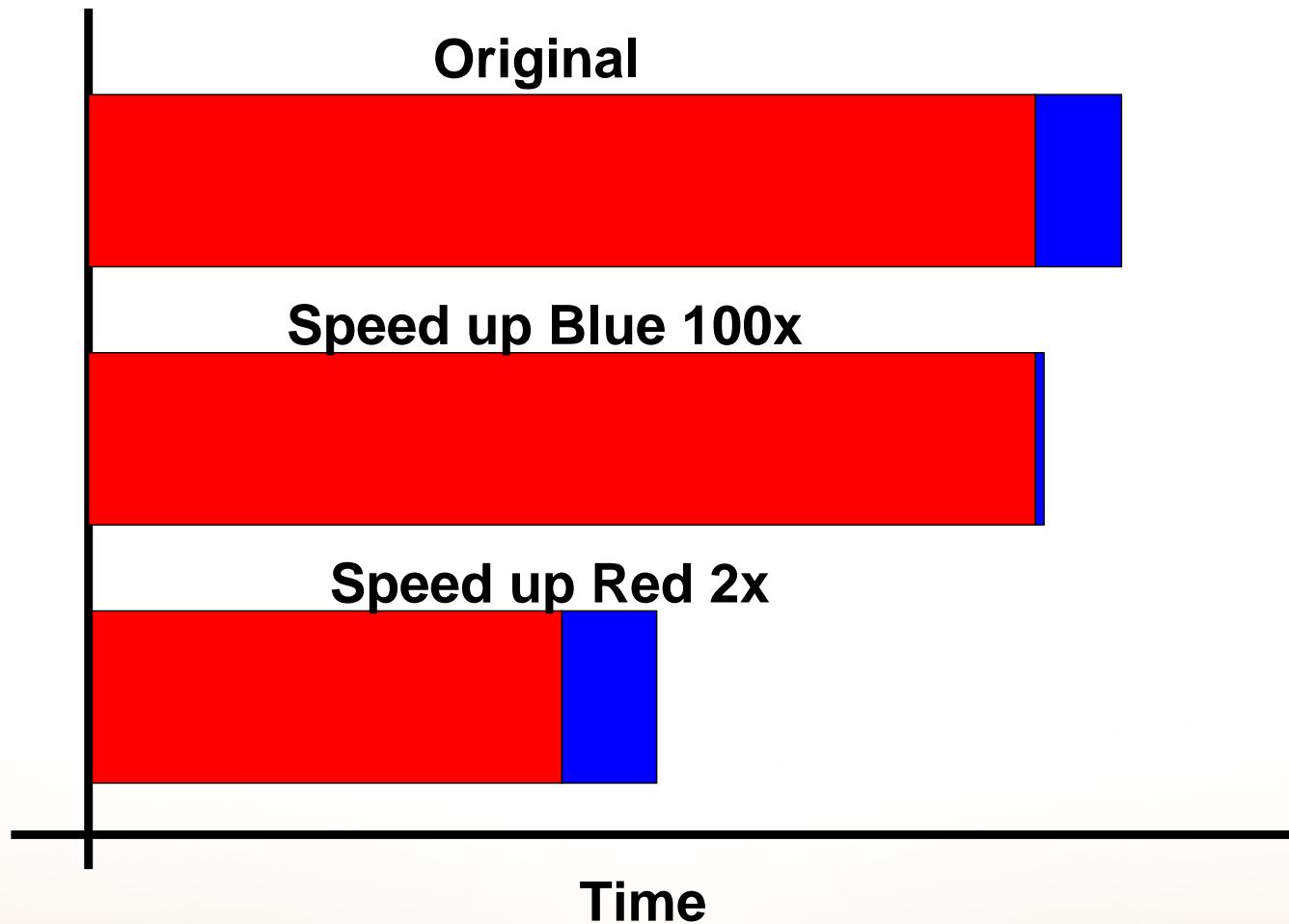
# Wisdom from Knuth

“We should forget about small efficiencies, say about 97% of the time:

**premature optimization is the root of all evil.**

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified” — Donald Knuth

# Amdahl's Law



# Profiling and Tracing

## Profiling

- Records aggregate performance metrics
- Number of times routine invoked
- Structure of invocations

## Tracing

- When and where events of interest took place
- Time-stamped events
- Shows when/where messages sent/received



# Profiling Details

- Records summary information during execution
- Usually Low Overhead
- Implemented via **Sampling** (execution periodically interrupted and measures what is happening) or **Measurement** (extra code inserted to take readings)



# Tracing Details

- Records information on significant events
- Provides timestamps for events
- Trace files are typically \*huge\*
- When doing multi-processor or multi-machine tracing, hard to line up timestamps

# Performance Data Analysis

## Manual Analysis

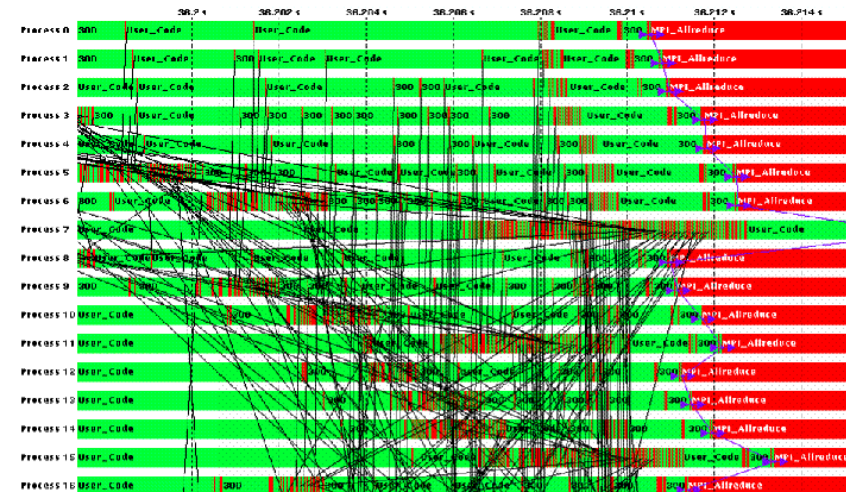
- Visualization, Interactive Exploration, Statistical Analysis
- Examples: TAU, Vampir

## Automatic Analysis

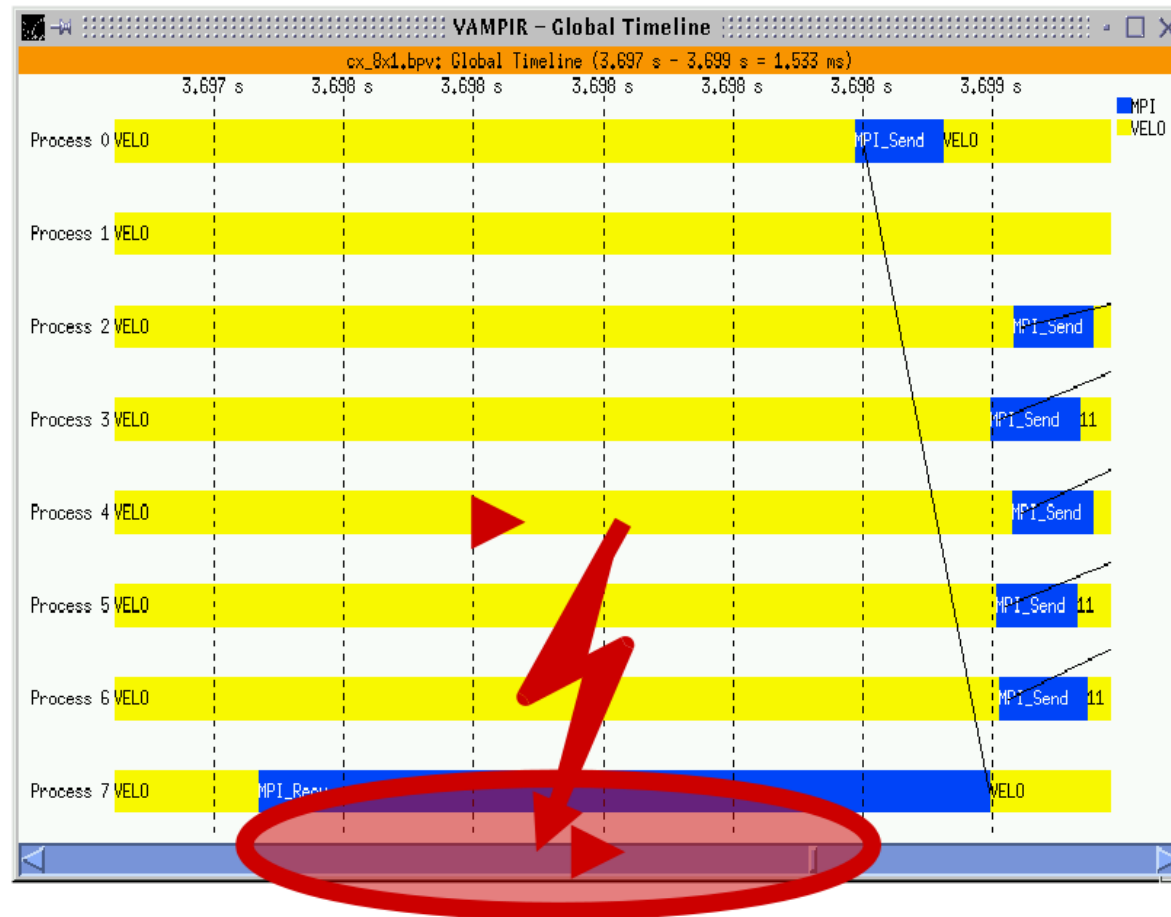
- Try to cope with huge amounts of data by automatic analysis
- Examples: Paradyn, KOJAK, Scalasca, Perf-expert

# Automated Performance Analysis

- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - ORNL's Jaguar Cray XT5: 224,256 compute cores (2010)
  - LLNL Sequoia (IBM, based on future Blue Gene arch.): ~1.6 million compute cores (2011-2012)
  - Trend to multi-core
- Large amounts of performance data
  - Several gigabytes or even terabytes
  - Overwhelms user
- Not all programmers are performance experts
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- Automation can solve some of these issues



# Automation Example



This is a situation that can be detected *automatically* by analyzing the trace file

-> **late sender** pattern

# PART IV — Hardware Performance Counters and PAPI



# Different Low-Level Tools on Linux

- Oprofile, Vtune — System Wide Sampled Profiling
- perf — Local and System-wide profiling as well as aggregate counts
- PAPI — Low-level library providing support for profiling as well as aggregate counting and self monitoring



# perf\_event on Linux

```
vweaver1:hydra10 ~> perf stat /bin/ls  
Performance counter stats for '/bin/ls':
```

1.202602	task-clock-msecs	#	0.119	CPUs
9	context-switches	#	0.007	M/sec
0	CPU-migrations	#	0.000	M/sec
281	page-faults	#	0.234	M/sec
3471913	cycles	#	2887.001	M/sec
1995325	instructions	#	0.575	IPC
40271	cache-references	#	33.487	M/sec
13670	cache-misses	#	11.367	M/sec
0.010071165 seconds time elapsed				

# The Performance API Library (PAPI)

- Low-level Crossplatform Performance Measurement Interface
- C, C++, Fortran (or attach to running process)
- Basis for more advanced visualization tools. Vampir, Tau, PerfExpert, etc.





# PAPI Features

- Provides high-level access to timers
- Provides high and low-level access to performance counters
- Provides profiling support
- Provides system information



# What are Hardware Performance Counters?

- Registers on CPU that measure low-level system performance
- Available on most modern CPUs; increasingly found on GPUs, network devices, etc.
- Low overhead to read

# Learning About the Counters

- Number of counters varies from machine to machine
- Available events different for every vendor and every generation
- Available documentation not very complete (Intel Vol3b, AMD BKDG)
- PAPI tries to hide this complexity from users

# Example Implementation – Hydra Lab Machines

- Nehalem Processor
- Counters: 4 general purpose, 3 fixed
- Events: 90 General (with many umasks), 3 Offcore, 44 Uncore, 26 Linux Software
- PEBS precise sampling and load latency monitoring
- LBR Last Branch recording

# papi\_avail

```
vweaver1:hydra10 ~> papi_avail | less
```

```
Available events and hardware information.
```

```
-----  
PAPI Version           : 4.1.3.0  
Vendor string and code : GenuineIntel (1)  
Model string and code  : Intel(R) Xeon(R) CPU X5550  
CPU Revision           : 5.000000  
CPUTID Info            : Family: 6  Model: 26  Stepping: 5  
CPU Megahertz           : 2666.822998  
CPU Clock Megahertz     : 2666  
Hdw Threads per core   : 1  
Cores per Socket       : 4  
Total CPU's            : 4  
Number Hardware Counters : 16  
Max Multiplex Counters  : 512
```

## papi\_avail continued

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
PAPI_L2_DCM	0x80000002	Yes	Yes	Level 2 data cache misses
PAPI_L2_ICM	0x80000003	Yes	No	Level 2 instruction cache
PAPI_L3_DCM	0x80000004	No	No	Level 3 data cache misses
PAPI_L3_ICM	0x80000005	No	No	Level 3 instruction cache
PAPI_L1_TCM	0x80000006	Yes	Yes	Level 1 cache misses
PAPI_L2_TCM	0x80000007	Yes	No	Level 2 cache misses
PAPI_L3_TCM	0x80000008	Yes	No	Level 3 cache misses

# Selected Native Events

UNHALTED\_CORE\_CYCLES

INSTRUCTIONS\_RETIRED

LAST\_LEVEL\_CACHE\_REFERENCES

LAST\_LEVEL\_CACHE\_MISSES

BRANCH\_INSTRUCTIONS\_RETIRED

DTLB\_LOAD\_MISSES

...

FP\_COMP\_OPS\_EXE:SSE\_FP\_PACKED

SQ\_MISC:FILL\_DROPPED



# papi\_native\_avail

```
vweaver1:hydra10 ~> papi_native_avail | less
```

```
Event Code    Symbol    | Long Description |
```

```
-----  
0x40000000    UNHALTED_CORE_CYCLES | count core clock cycles  
              whenever the clock signal on the specific core  
              is running (not halted).  
-----
```

```
0x40000001    INSTRUCTION_RETIRED | count the number of  
              instructions at retirement.  
-----
```

```
0x40000004    LLC_REFERENCES | count each request originating  
              from the core to reference a cache line in the  
              last level cache. The count may include  
              speculation, but excludes cache line fills due  
              to hardware prefetch.
```



# Code Example

```
int main(int argc, char** argv) {  
  
    matrix_multiply(n, A, B, C);  
  
    return 0;  
}
```

# PAPI Timers

```
/* Compile with gcc -O2 -Wall mmm_papi_timer.c -lpapi */
#include <papi.h>

int main(int argc, char** argv) {

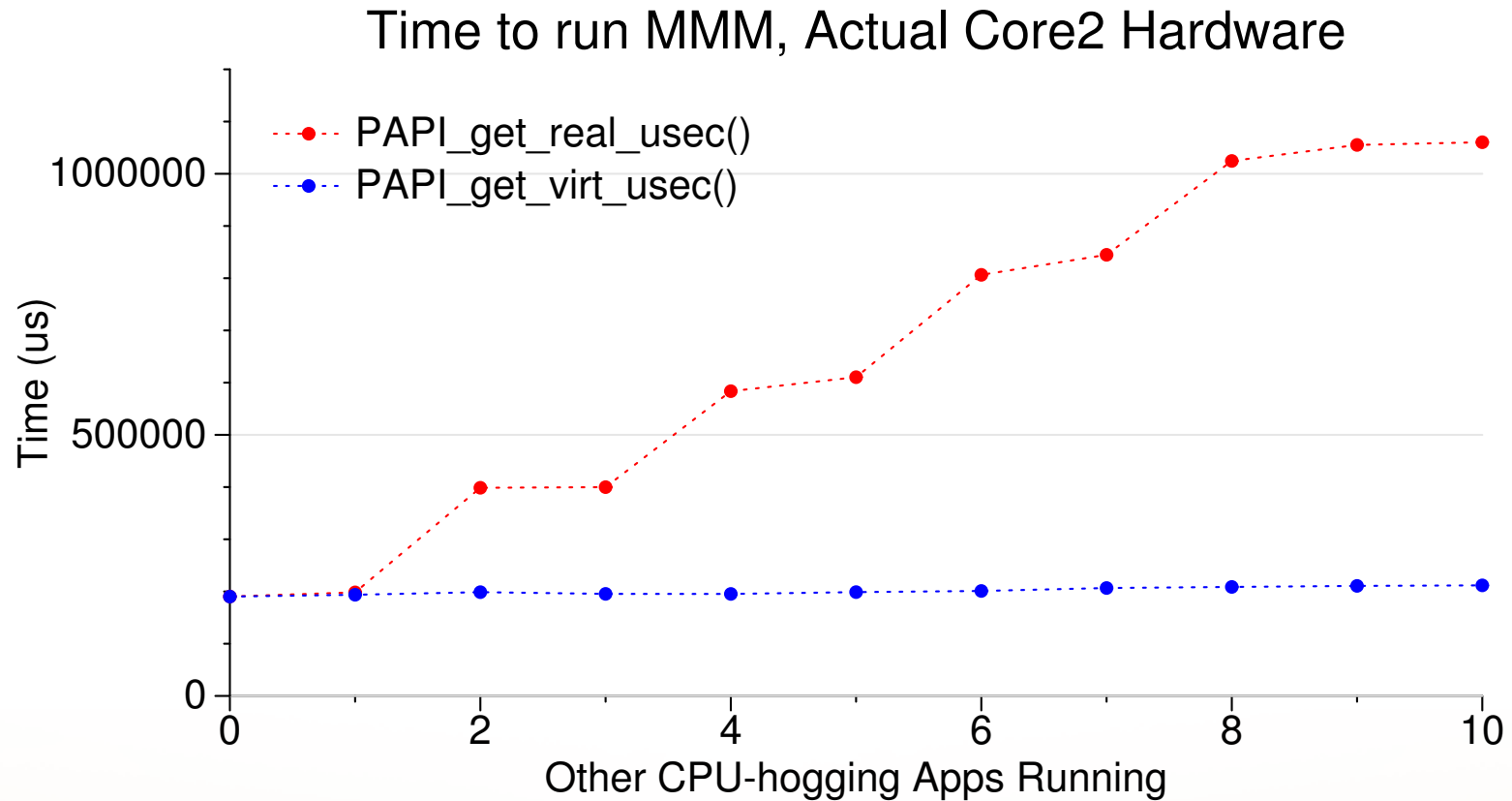
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI_library_init_␣%d␣\n", retval);
    }
    start_usecs=PAPI_get_real_usec();
    matrix_multiply(n, A, B, C);
    stop_usecs=PAPI_get_real_usec();
    printf("Elapsed_␣usecs=%d␣\n", stop_usecs - start_usecs);
    return 0;
}
```



# PAPI\_get\_real\_usec() vs PAPI\_get\_virt\_usec()

- PAPI\_get\_real\_usec()  
wall-clock time  
maps to `clock_gettime(CLOCK_REALTIME)`
- PAPI\_get\_virt\_usec()  
only time process is actually running  
maps to `clock_gettime(CLOCK_THREAD_CPUTIME_ID)`

# Real vs Virt Timer Results



# Don't forget the man pages!

```
vweaver1:hydra10 ~> man PAPI_get_real_usec  
PAPI_get_real_cyc(3)          PAPI
```

## NAME

PAPI\_get\_real\_cyc - get real time counter in clock cycles

PAPI\_get\_real\_usec - get real time counter in microseconds

## DESCRIPTION

Both of these functions return the total real time passed since some arbitrary starting point. The time is returned in clock cycles or microseconds respectively. These calls are equivalent to wall clock time.

# Measuring Floating Point Usage

- We'll use the PAPI\_FP\_OPS pre-defined counter
- On Nehalem this maps to  
 $\text{FP\_COMP\_OPS\_EXE:SSE\_FP} + \text{FP\_COMP\_OPS\_EXE:X87}$

# PAPI FP Measurement

```
#include <papi.h>

int main(int argc, char** argv) {

    long long values[1];
    int event_set=PAPI_NULL;
    int ntv;

    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI_library_init\n");
    }
}
```

# PAPI FP Measurement Continued

```
retval = PAPI_event_name_to_code( "PAPI_FP_OPS", &ntv );  
if (retval != PAPI_OK) {  
    fprintf(stderr, "Error converting PAPI_FP_OPS\n");  
}  
  
retval = PAPI_create_eventset( &event_set );  
if (retval != PAPI_OK) {  
    fprintf(stderr, "Error creating event_set\n");  
}  
  
retval = PAPI_add_event( event_set, ntv );  
if (retval != PAPI_OK) {  
    fprintf(stderr, "Error adding PAPI_FP_OPS\n");  
}
```





# PAPI FP Measurement Continued

```
retval = PAPI_start( event_set );  
  
matrix_multiply(n, A, B, C);  
  
retval = PAPI_stop( event_set, values );  
  
printf("PAPI_FP_OPS = %lld\n", values[0]);  
  
return 0;  
}
```

# Results

```
vweaver1:hydra10 ~/class/cs340/src> ./mmm_papi_fp  
PAPI_FP_OPS = 2012334
```

- Code: naive 100x100 matrix-matrix multiply.
- Expected value: 100x100 multiplies + 100x100 additions  
= 2,000,000 floating point operations.
- The result 2,012,334 seems reasonable.

# GigaFLOP/s

A more interesting benchmark might be measuring how many GigaFLOP/s (Billions of Floating Point Operations Per Second) the code can sustain.

# Code modified to measure GFLOP/s

```
retval = PAPI_start( event_set );
start_usecs=PAPI_get_virt_usec();

matrix_multiply(n, A, B, C);

stop_usecs=PAPI_get_virt_usec();
retval = PAPI_stop( event_set, values );

flops = ((double)values[0]/
         (double)(stop_usecs-start_usecs))*1.0e6;

printf("GigaFLOP/s = %.3f\n", flops/1.0e9);
```

# GFLOP/s Results

```
vweaver1:hydra10 ~/class/cs340/src> ./mmm_papi_gflops  
GigaFLOP/s = 1.667
```

Theoretical Peak performance of a Xeon 5500:

85.12 GFLOP/s.

There is still some optimization that can be done here.

## GFLOPS Results continued

Theoretical Peak performance of a Xeon 5500:

85.12 GFLOP/s.

Why code it yourself if a library is available?

implementation	usecs	FLOPs	GFLOP/s
Naive	1200	2,000,635	1.667
ATLAS	560	1,037,425	1.853
GOTO	305	254,758	0.835

# Parallel Code

- If using pthreads, have to enable it in PAPI
- Enable “inherit” to get combined stats for all subthreads

# Cycles Per Instruction (CPI)

Most modern CPUs are super-scalar, meaning they can execute multiple instructions per cycle (this is unrelated to, but complements, multi-core). They can usually also execute instructions out-of-order but that's another story. CPI measures how much intra-thread parallelization is happening.



# Measuring CPI

- We'll use the PAPI\_TOT\_CYC and PAPI\_TOT\_INS pre-defined counters
- On Nehalem these map to UNHALTED\_CORE\_CYCLES and INSTRUCTION\_RETIRED

# Code modified to measure CPI

```
long long values[2];  
  
retval = PAPI_create_eventset( &event_set );  
  
retval = PAPI_event_name_to_code("PAPI_TOT_CYC", &ntv);  
retval = PAPI_add_event( event_set, ntv );  
  
retval = PAPI_event_name_to_code("PAPI_TOT_INS", &ntv);  
retval = PAPI_add_event( event_set, ntv );
```

# Code modified to measure CPI

```
retval = PAPI_start( event_set );  
  
matrix_multiply(n, A, B, C);  
  
retval = PAPI_stop( event_set, values );  
  
printf("Cycles_=%lld\n", values[0]);  
printf("Retired_Instructions_=%lld\n", values[1]);  
printf("CPI_=%.2f\n", (double) values[0] /  
                      (double) values[1]);  
printf("IPC_=%.2f\n", (double) values[1] /  
                      (double) values[0]);
```

# CPI Results

```
vweaver1:hydra10 ~/class/cs340/src> ./mmm_papi_cpi  
Cycles = 3418330  
Retired Instructions = 8112910  
CPI = 0.42  
IPC = 2.37
```

Theoretical Peak performance of a Xeon 5500:

CPI = 0.25, IPC = 4.0

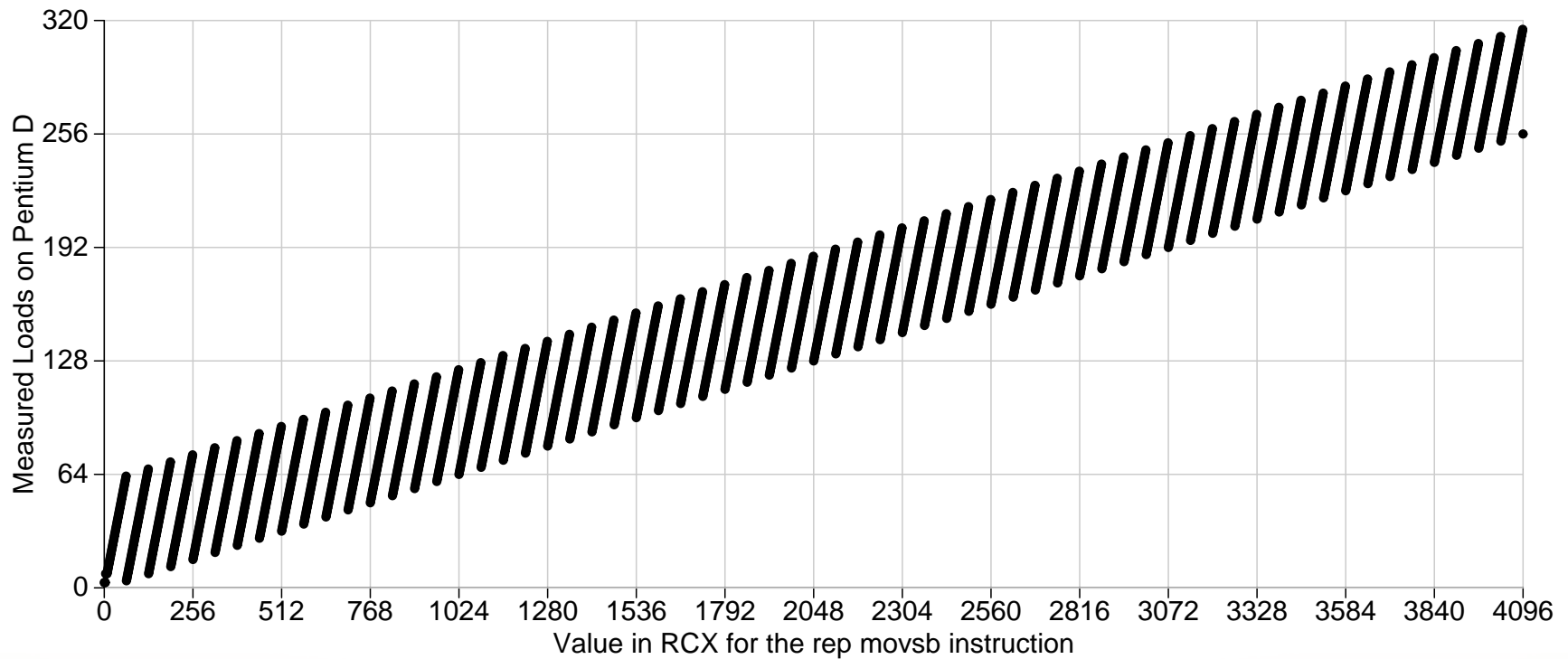
# Other Metrics

- Branch Mispredictions: PAPI\_BR\_MSP
- Cache Misses: PAPI\_L2\_TCM
- Resource Stalls: PAPI\_STL\_ICY
- TLB Misses: PAPI\_TLB\_TL

# Cautions when Using the Counters

- Not-well Validated
- Low-priority CPU feature
- Not-well Documented

# Pentium 4 Retired Stores Result



# Non-CPU Counters

- GPU Counters
- Network, Infiniband
- Disk I/O
- Power, Energy, Temperature



# Energy/Thermal, Low-End



# Energy/Thermal, High-End



- Thousands of processors: every Joule counts
- Running the Air Conditioning could cost more than running the machine

# PAPI-V Issues

- Time?
- Exporting VM Statistics?
- Virtualized Counters

# Some Tools that Use PAPI

- **TAU** (U Oregon) <http://www.cs.uoregon.edu/research/tau/>
- PerfSuite (NCSA) <http://perfsuite.ncsa.uiuc.edu/>
- HPCToolkit (Rice Univ) <http://hipersoft.cs.rice.edu/hpctoolkit/>
- **KOJAK and SCALASCA** (FZ Juelich, UTK) <http://icl.cs.utk.edu/kojak/>
- **VampirTrace and Vampir** (TU Dresden) <http://www.vamir.eu>
- Open|Speedshop (SGI) <http://oss.sgi.com/projects/openspeedshop/>
- SvPablo (UNC Renaissance Computing Institute) <http://www.renci.unc.edu/Software/Pablo/pablo.htm>
- ompP (UTK) <http://www.ompp-tool.com>



# PART V — High Level Tools

# VAMPIR: Performance Analysis Suite

Consists of:

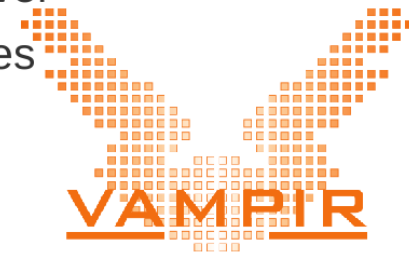
- VampirTrace part for instrumentation, monitoring and recording
- VampirServer part for visualization and analysis

## VampirTrace

- Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

## VampirServer

- Implements client / server model with distributed server
- Allows very scalable interactive visualization for traces

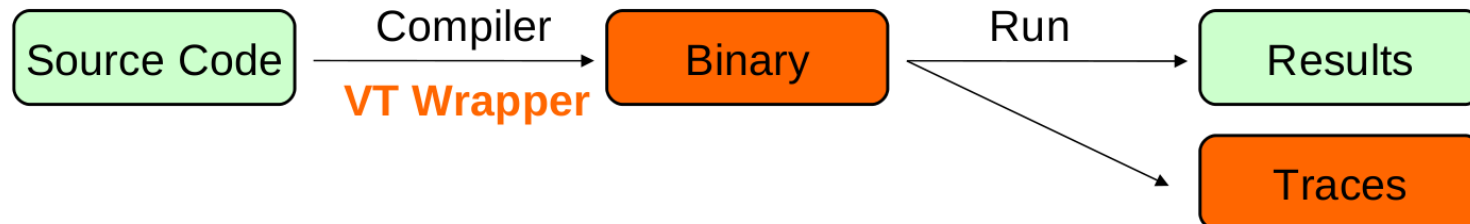


# Instrumentation with VAMPIRTRACE

Edit – Compile – Run Cycle



Edit – Compile – Run Cycle with VampirTrace



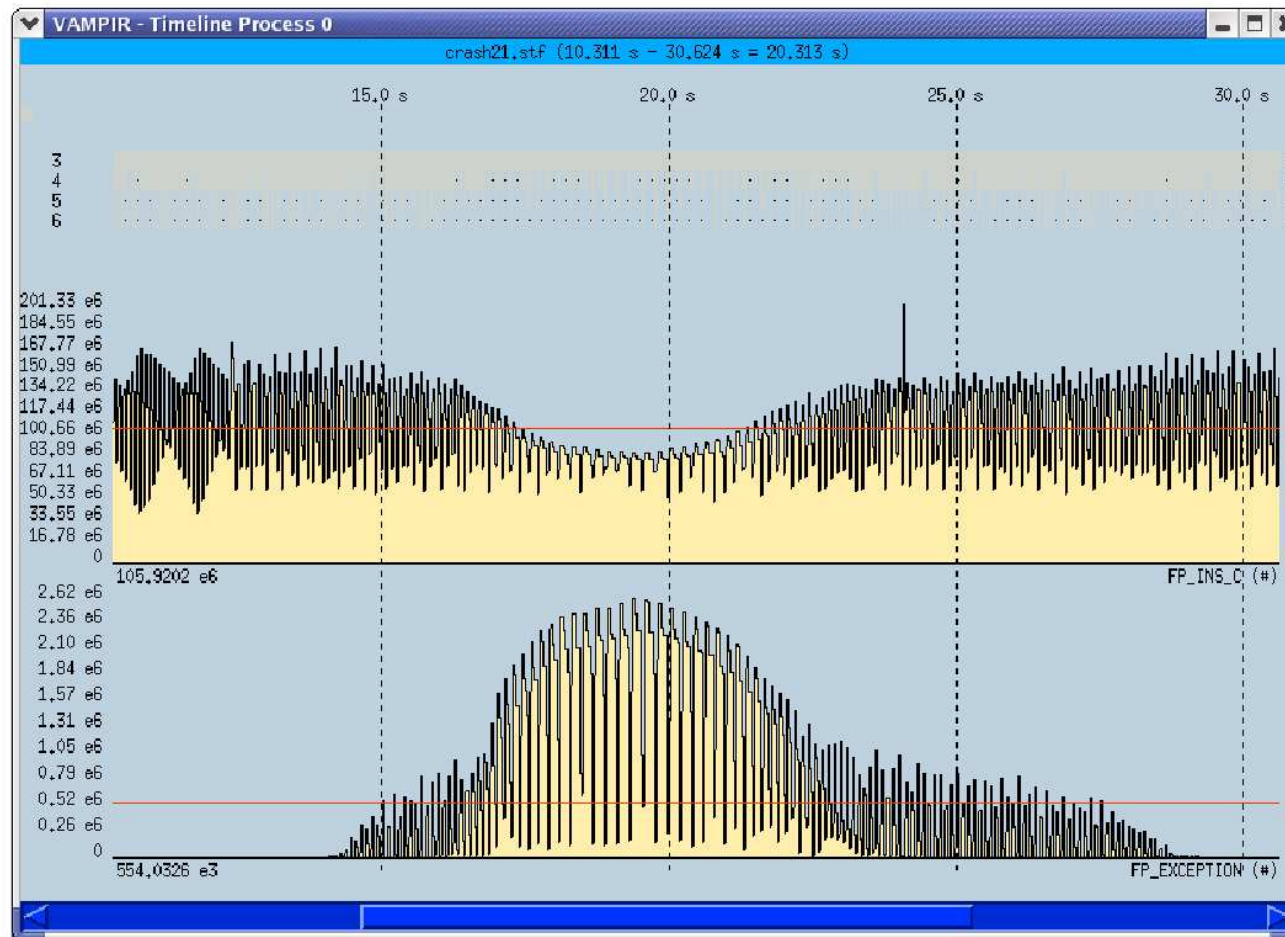


# VAMPIR Example



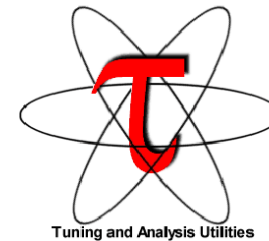


# PAPI FP Exception Example



# Tau Parallel Performance System

- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Integration in complex software, systems, applications

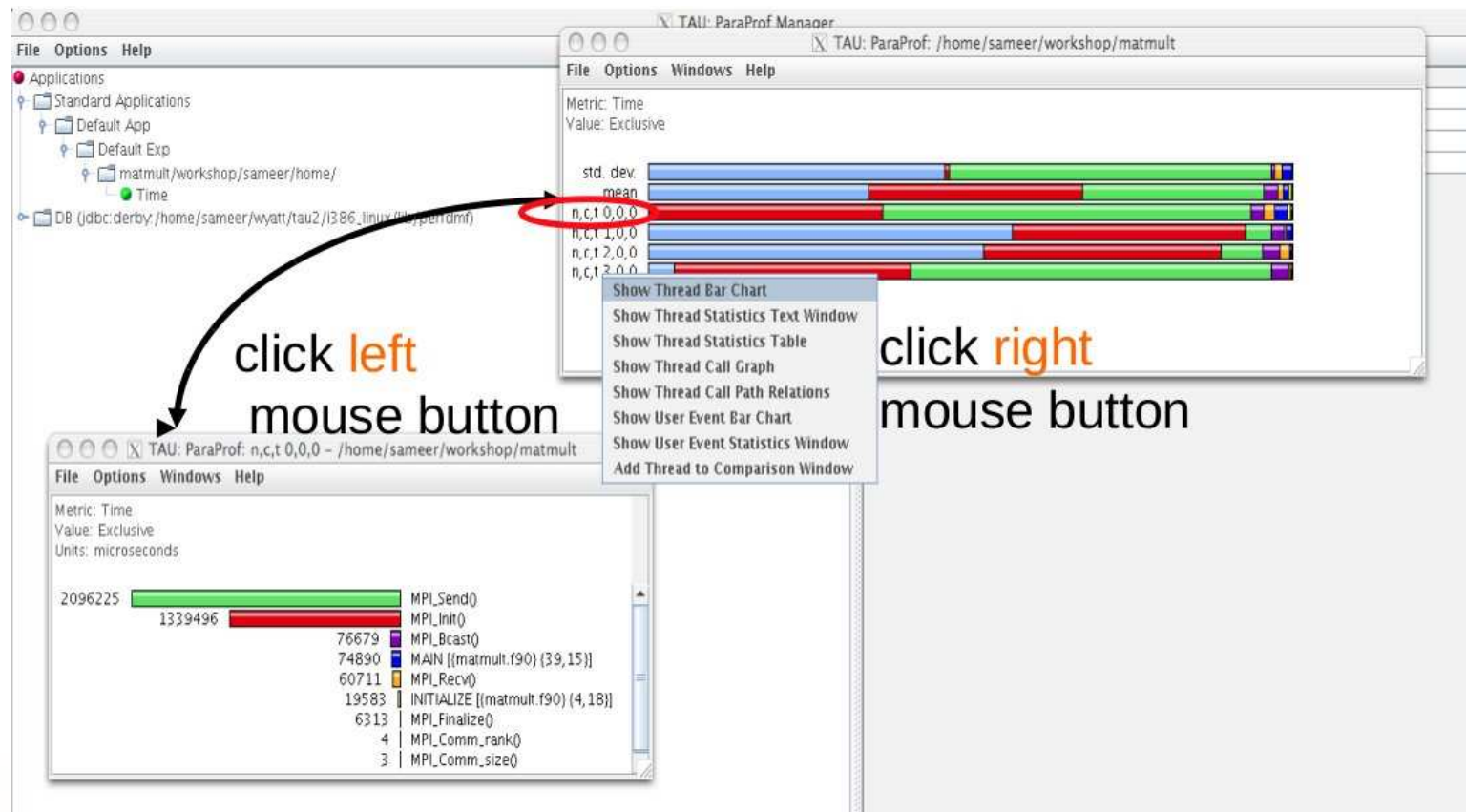


# Tau Instrumentation

Flexible instrumentation mechanisms at multiple levels:

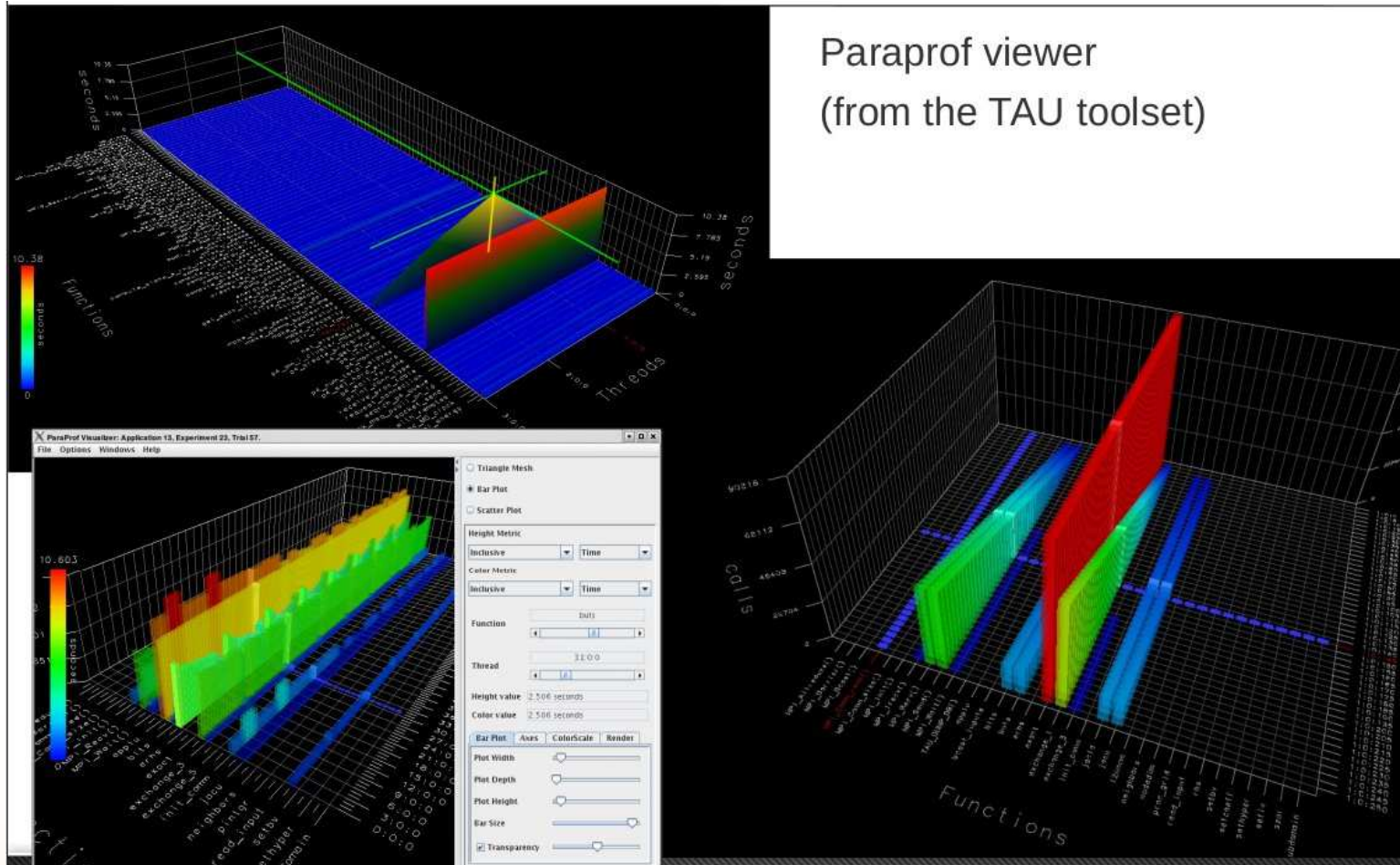
- **Source code**
  - manual (TAU API, TAU Component API)
  - automatic
    - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
    - OpenMP (directive rewriting (*Opari*), *POMP* spec)
- **Object code**
  - pre-instrumented libraries (e.g., MPI using *PMPI*)
  - statically-linked and dynamically-linked
- **Executable code**
  - dynamic instrumentation (pre-execution) (*DynInstAPI*)
  - virtual machine instrumentation (e.g., Java using *JVMPI*)
  - Python interpreter based instrumentation at runtime

# Tau Example



# Tau Example

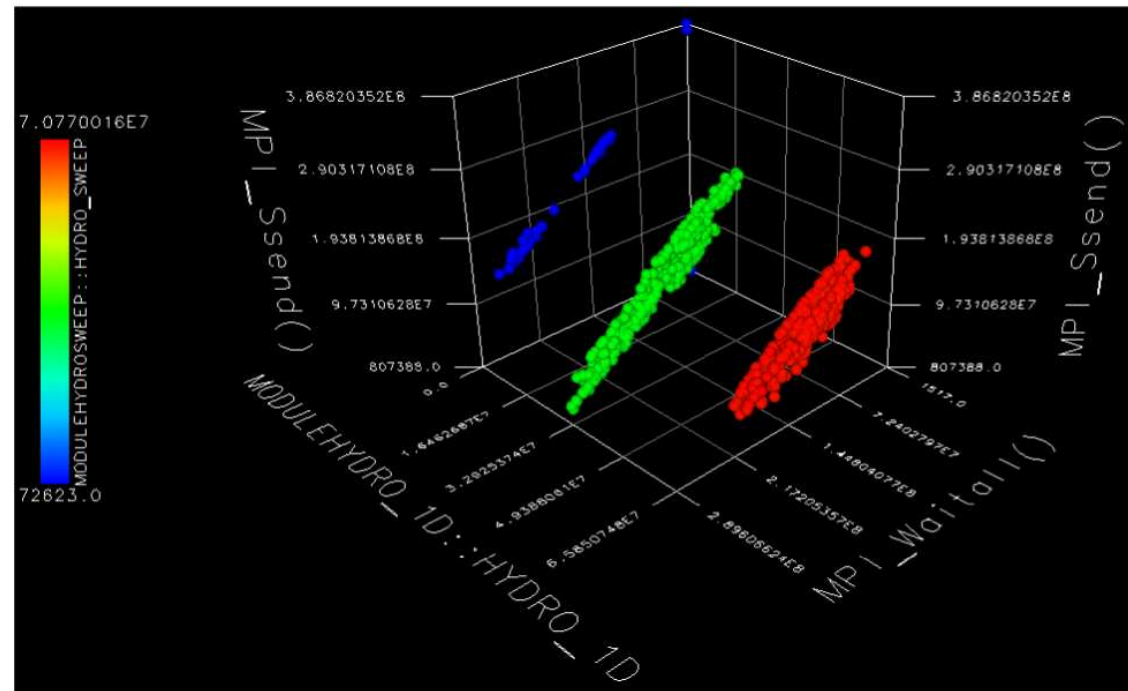
Paraprof viewer  
(from the TAU toolset)





# Tau Example

- Four significant events automatically selected (from 16K processors)
- Clusters and correlations are visible



# Conclusions

- Performance Analysis is important in HPC
- Performance Analysis tools are available that can ease optimization

# Questions?