
Parallelizing Atlas Reconstruction and Simulation: Issues and Optimization Solutions for Scaling on Multi- and Many-CPU Platforms

Charles Leggett¹

Sebastien Binet², Paolo Calafiura¹, Keith Jackson¹, David
Levinthal³, Mous Tatar khanov¹, Yushu Yao¹

¹Lawrence Berkeley National Lab, ²LAL, ³Intel

Introduction

- Days of easy performance gains from faster CPU frequency are over.
 - CPU manufacturers are putting multiple cores on each CPU
 - Hyperthreading increase number of virtual cores
- Atlas reconstruction uses ~ 1.5Gb of memory, so we will run out of memory if we just run multiple jobs simultaneously
- We will need to parallelize our code
- Two foci
 - The results of parallelizing Atlas reconstruction, and how it scales with the number of cores
 - The issues we discovered while measuring scaling, and the implications

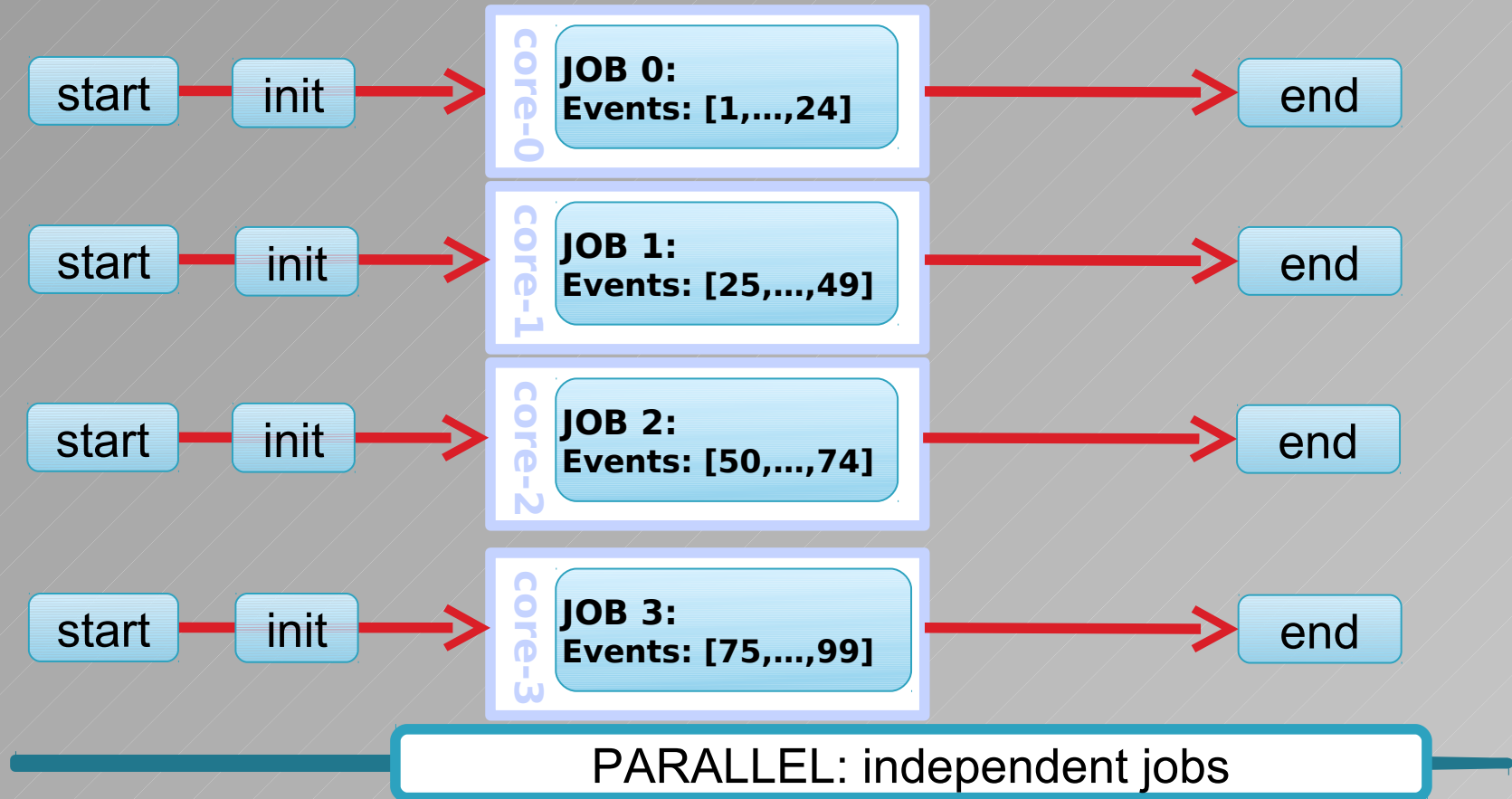
Styles of Parallelism

- Job
 - Easiest
 - Least data/code sharing
 - No code rewriting
 - AthenaMJ
- Event
 - Share configuration information, services, some code
 - Code changes to framework, transparent to users
 - Parallelize I/O
 - AthenaMP
- Sub-event
 - Pick “regions of interest” within event to parallelize, eg calorimeter
- Algorithmic
 - True vectorization
 - Hardest
 - Requires much re-writing of user code

Job Level Parallelism with AthenaMJ

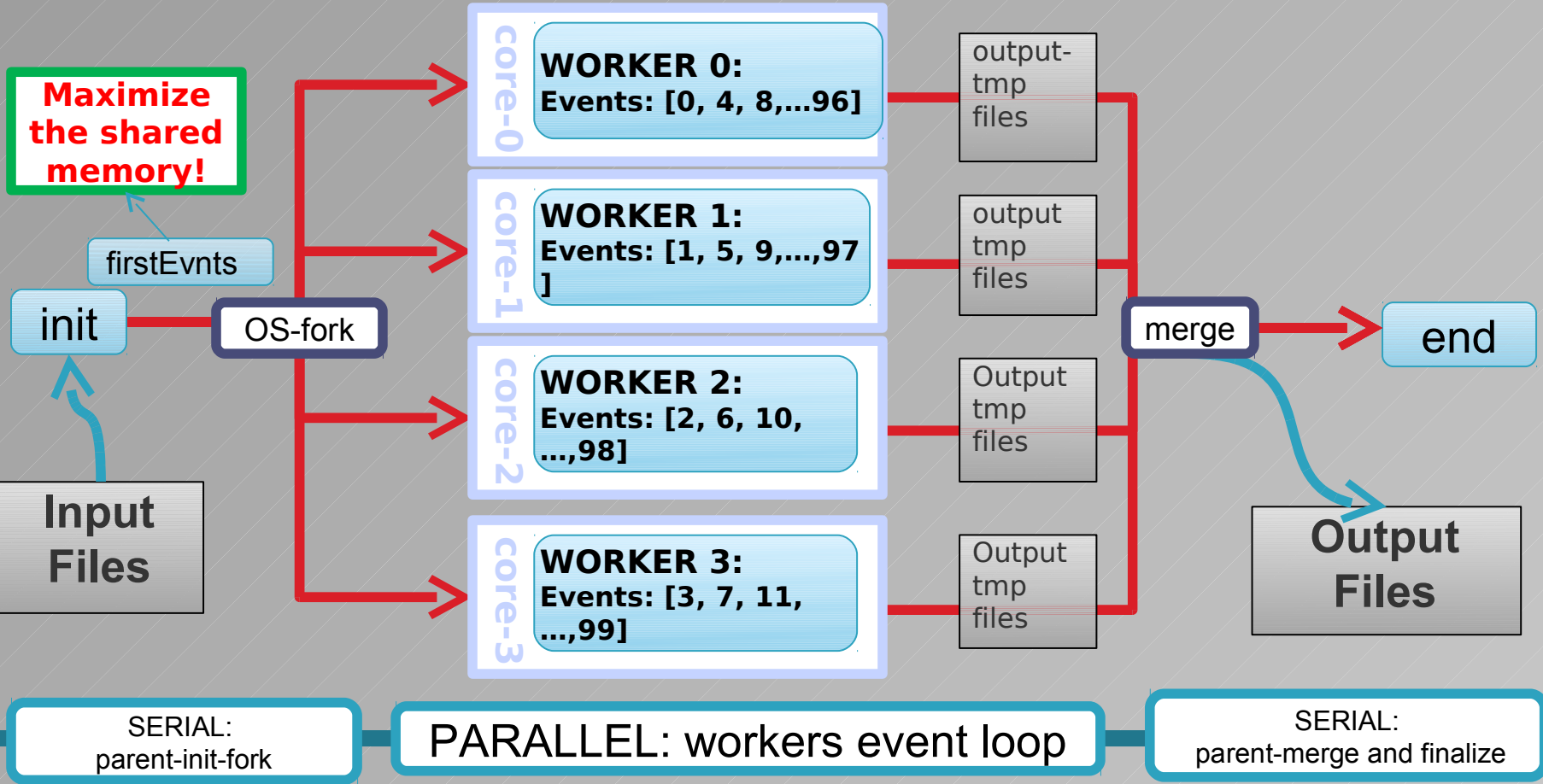


```
for i in range(4):  
    $> Athena.py -c "EvtMax=25; SkipEvents=$i*25" Job0.py
```



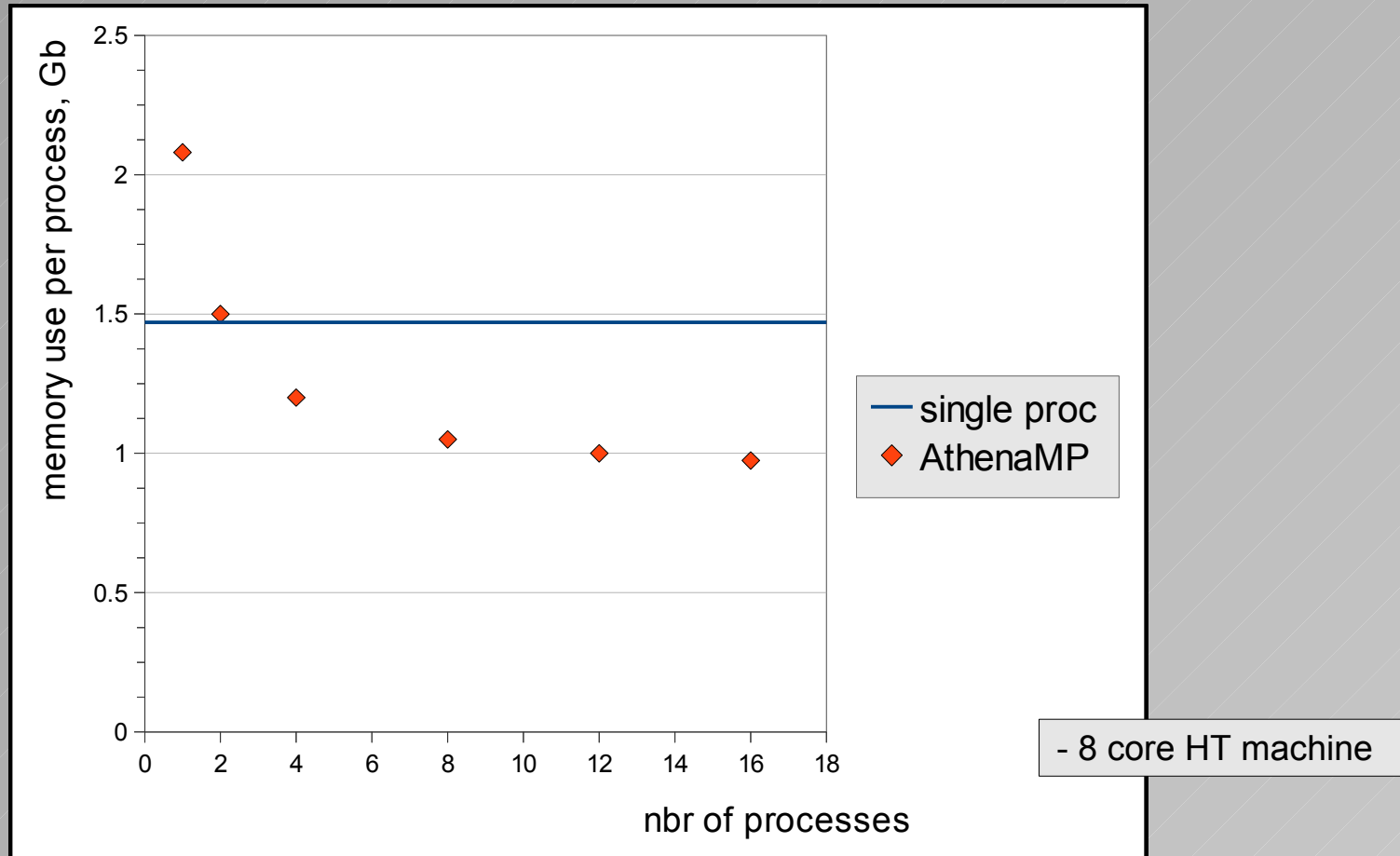
Event Level Parallelism with AthenaMP

```
> Athena.py --nprocs=4 -c EvtMax=100 Jobo.py
```



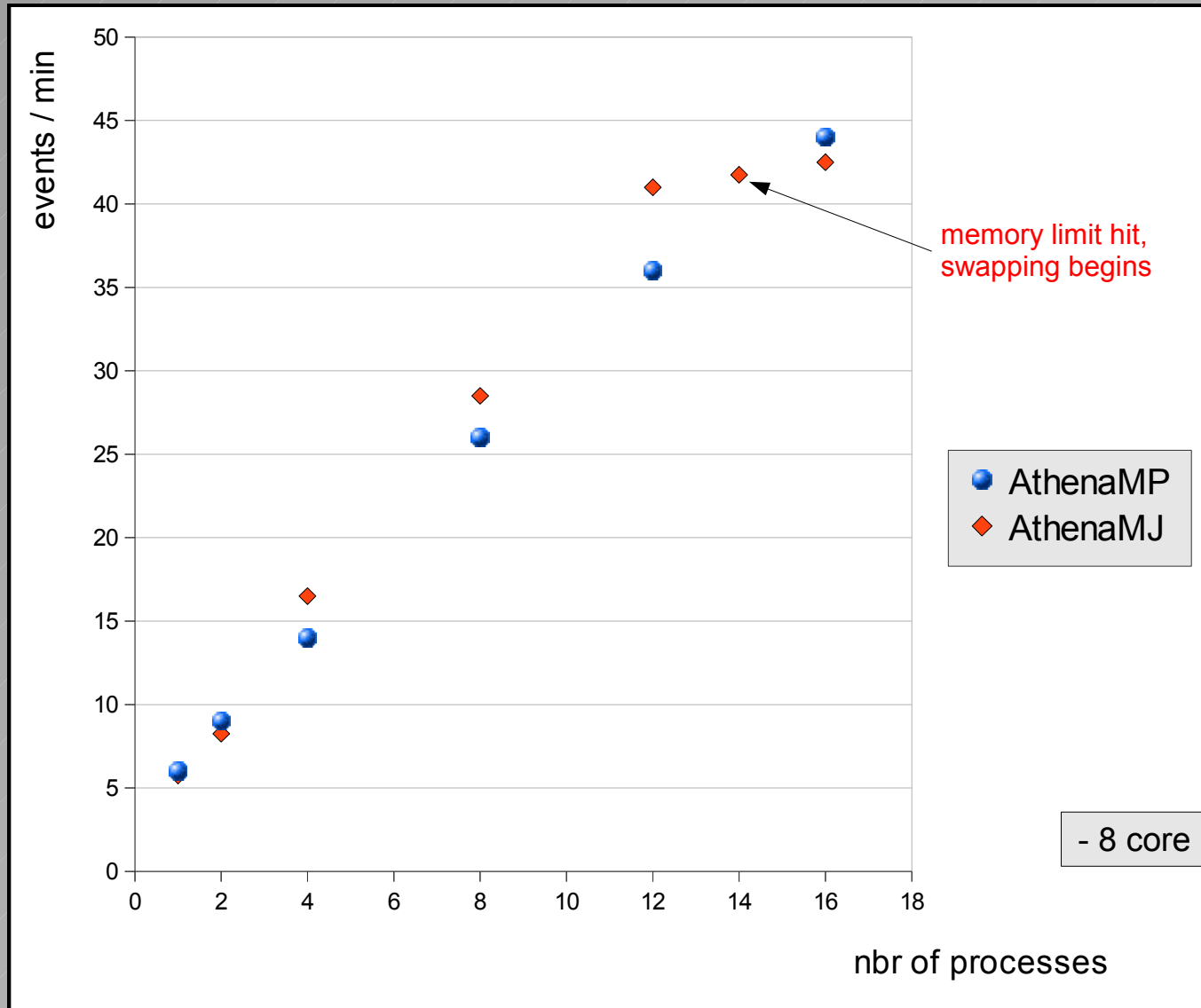
Memory Usage

- Single reco process uses ~1.5 Gb
- We can save significant memory through OS level sharing



AthenaMP ~0.5 Gb physical memory saved per process

Event Throughput



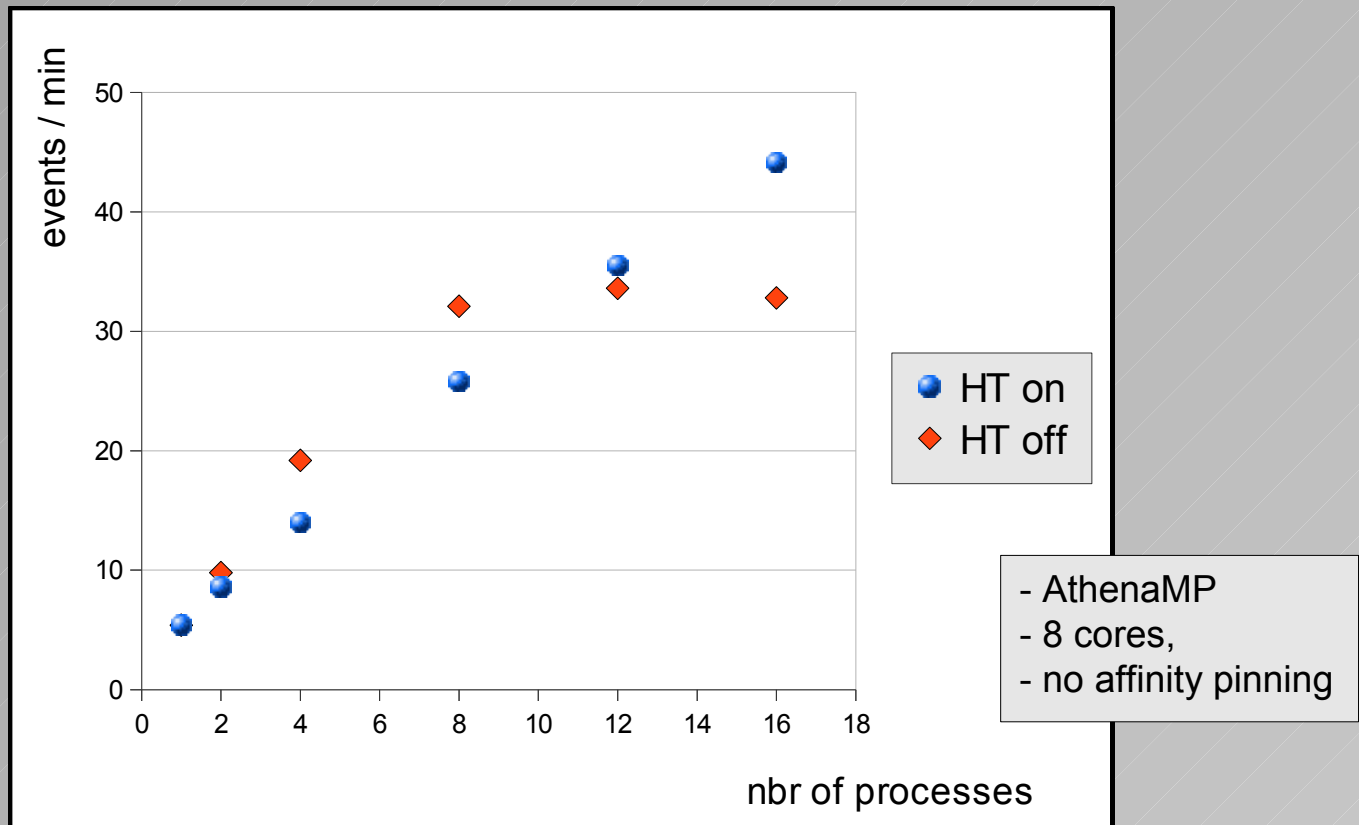
memory limit hit,
swapping begins

● AthenaMP
◆ AthenaMJ

- 8 core HT machine

Hyper-Threading

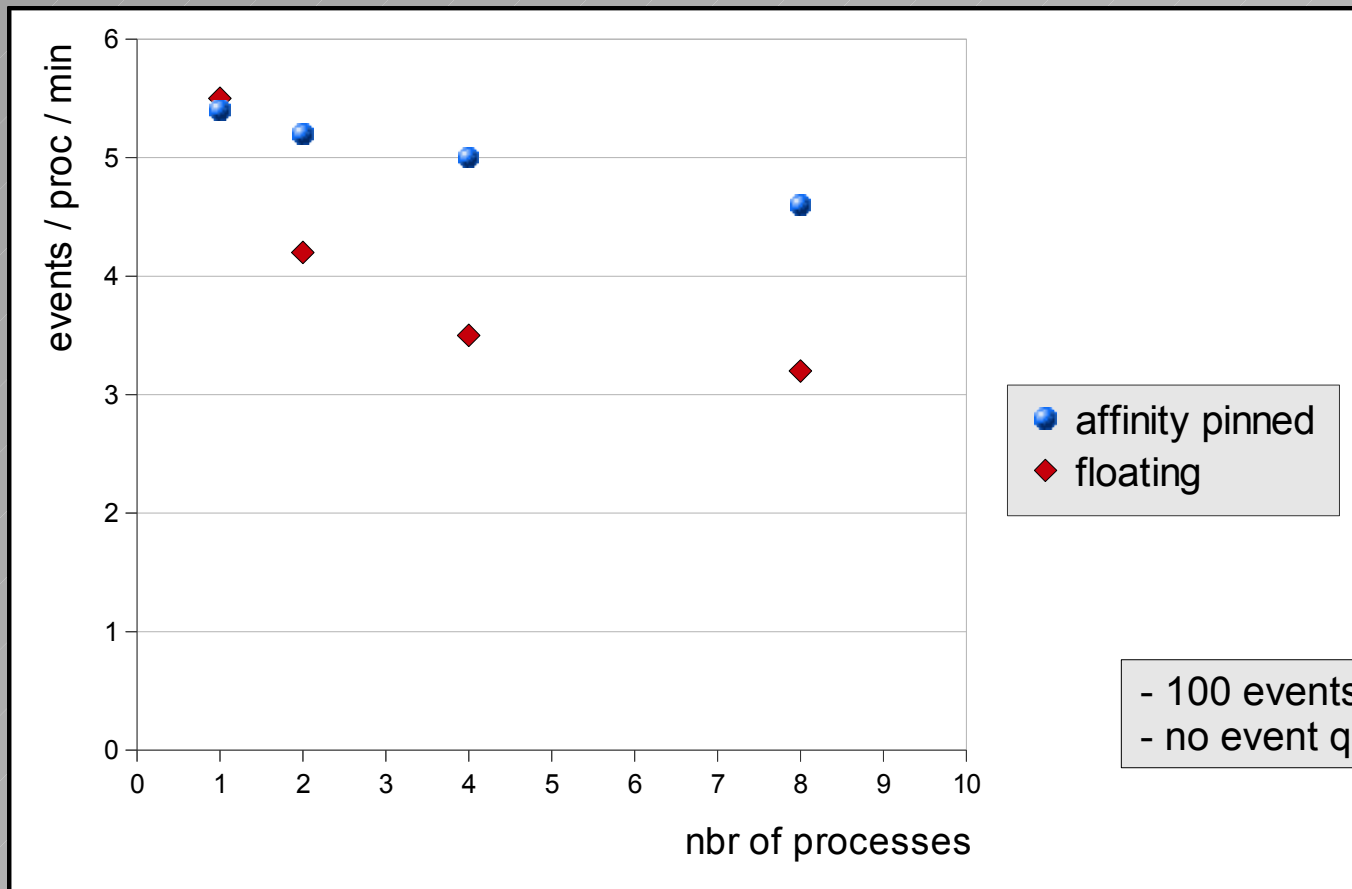
- Nature of instruction pipeline allows instructions to be interleaved
 - OS sees “virtual cores” as just another processor
 - Only effective until one of the threads saturates a shared resource and stalls



- Turn on HT when you have more processes than cores

CPU Affinity

- Linux schedulers will move processes from core to core during the course of a job
- We can prevent this by pinning a process to a core via its affinity, and gain 20%



- 100 events
- no event queue

Part II

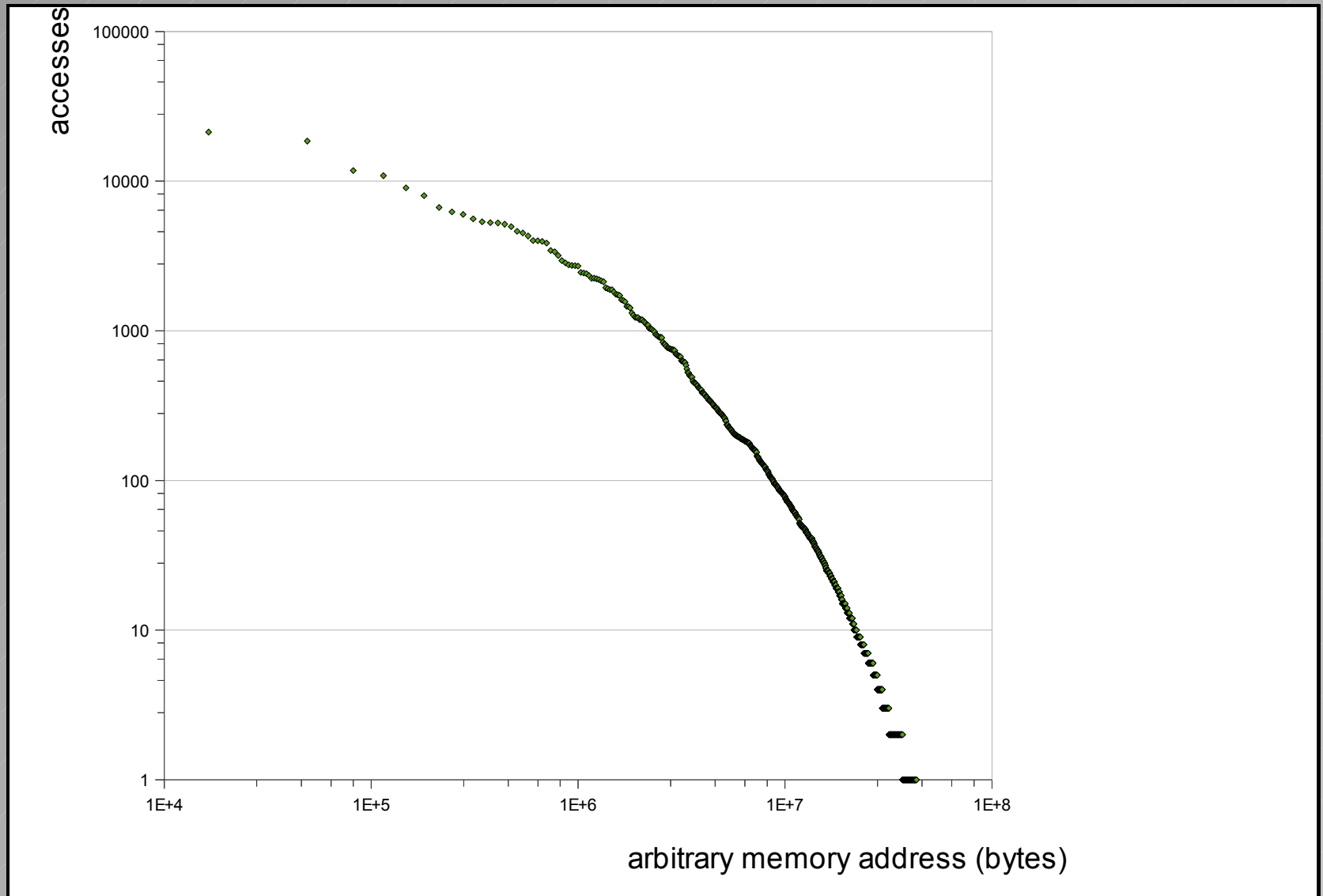
- Tools used to study performance and direct optimization
- Discoveries made from studying hardware level events

- Linux tools:
 - sar (I/O to disk and system loads)
 - vmstat: memory performance
 - IPM: time spent in I/O vs computation
 - numastat/numactl: reports/controls NUMA memory settings
- Intel Performance Tuning Utility (**PTU**)
 - Uses linux kernel module to provide a sampling profiler
 - Captures information from hardware counters available on Intel chips
 - Information available varies between processor families
 - Most accurate tool to understand what's going on at the hardware level
 - HUGE number of counters available
 - Need an expert to know which counters to profile

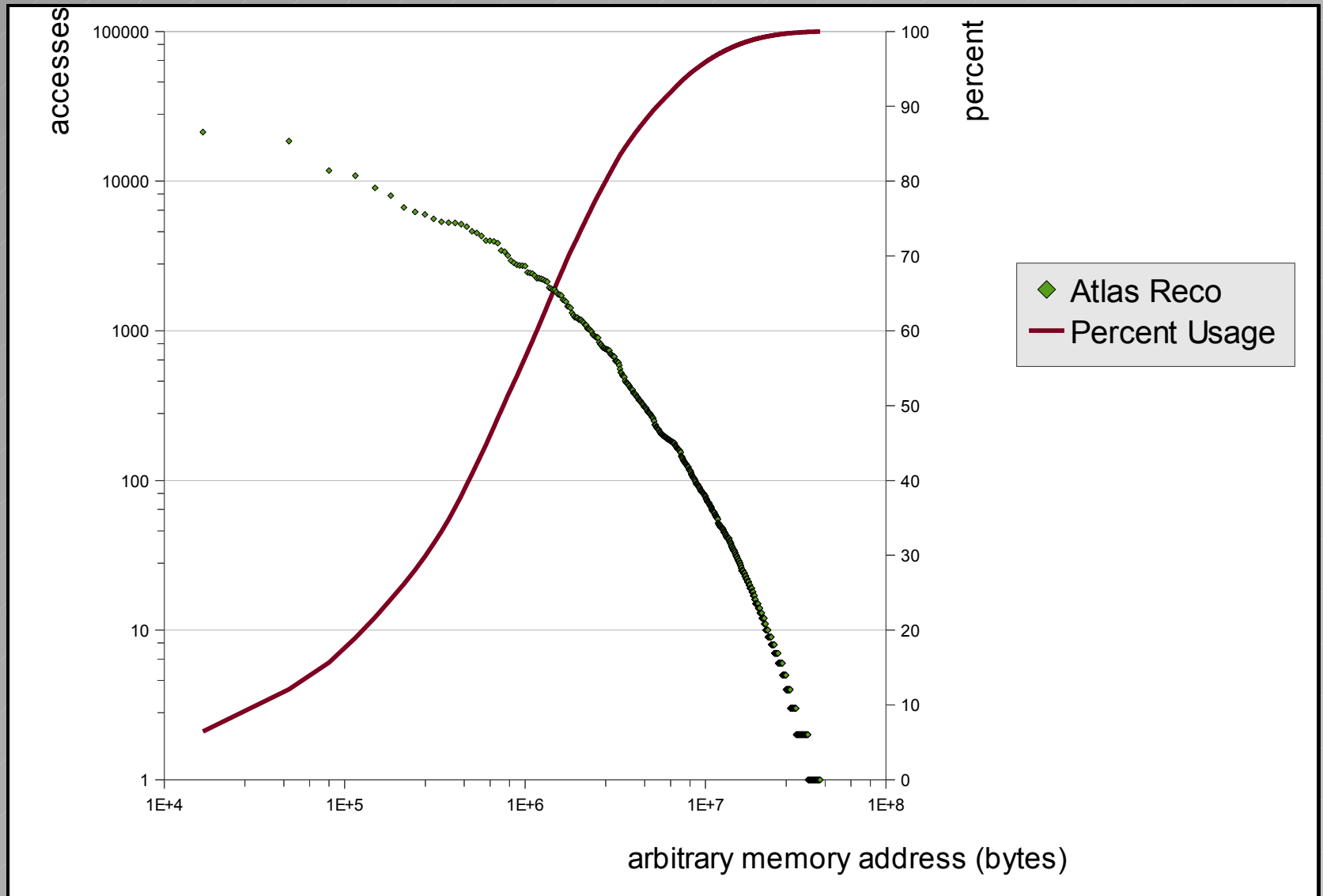
Initial Assumptions

- Our initial assumption was that we were I/O and memory bandwidth limited
 - We were WRONG

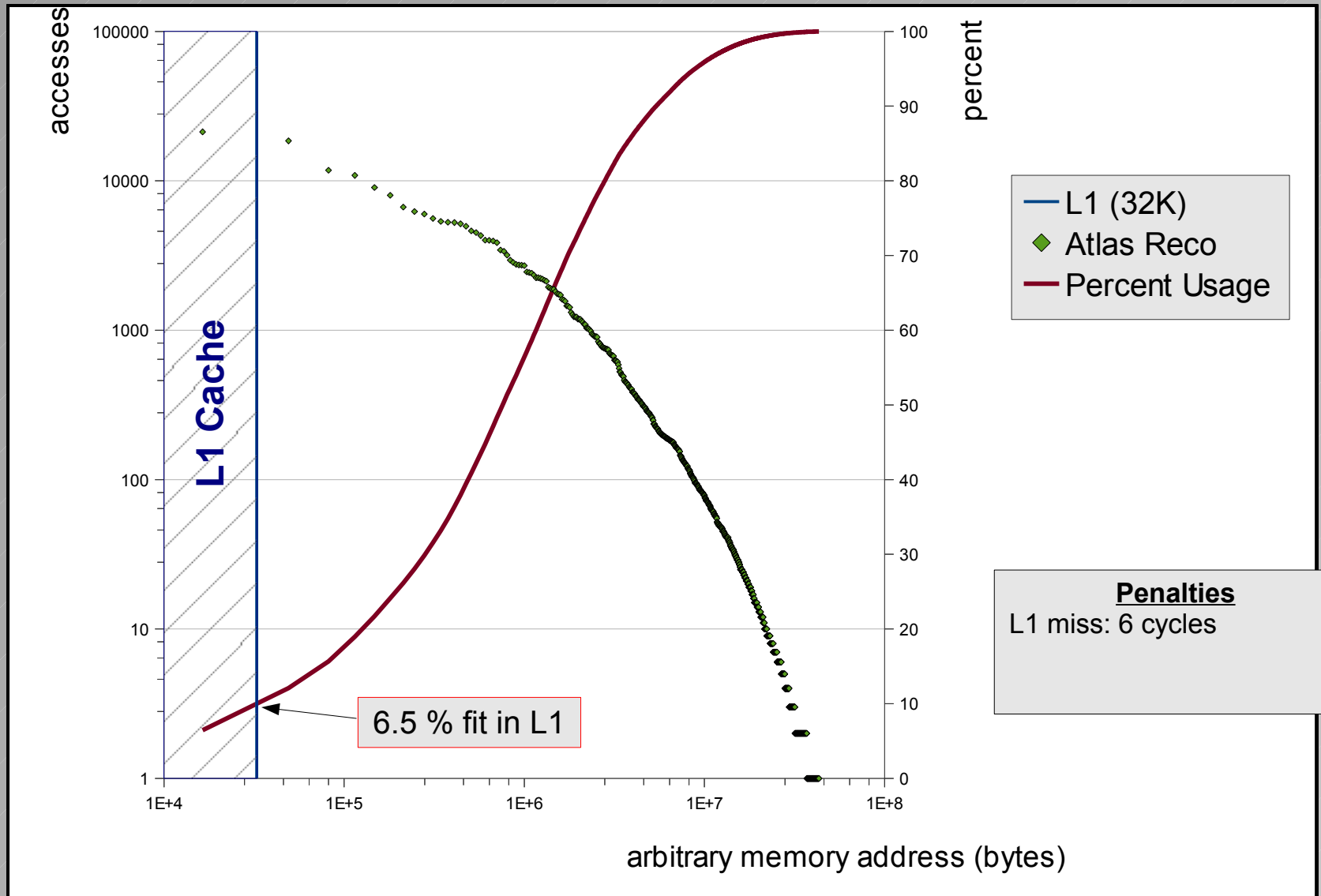
Size of Atlas Reconstruction



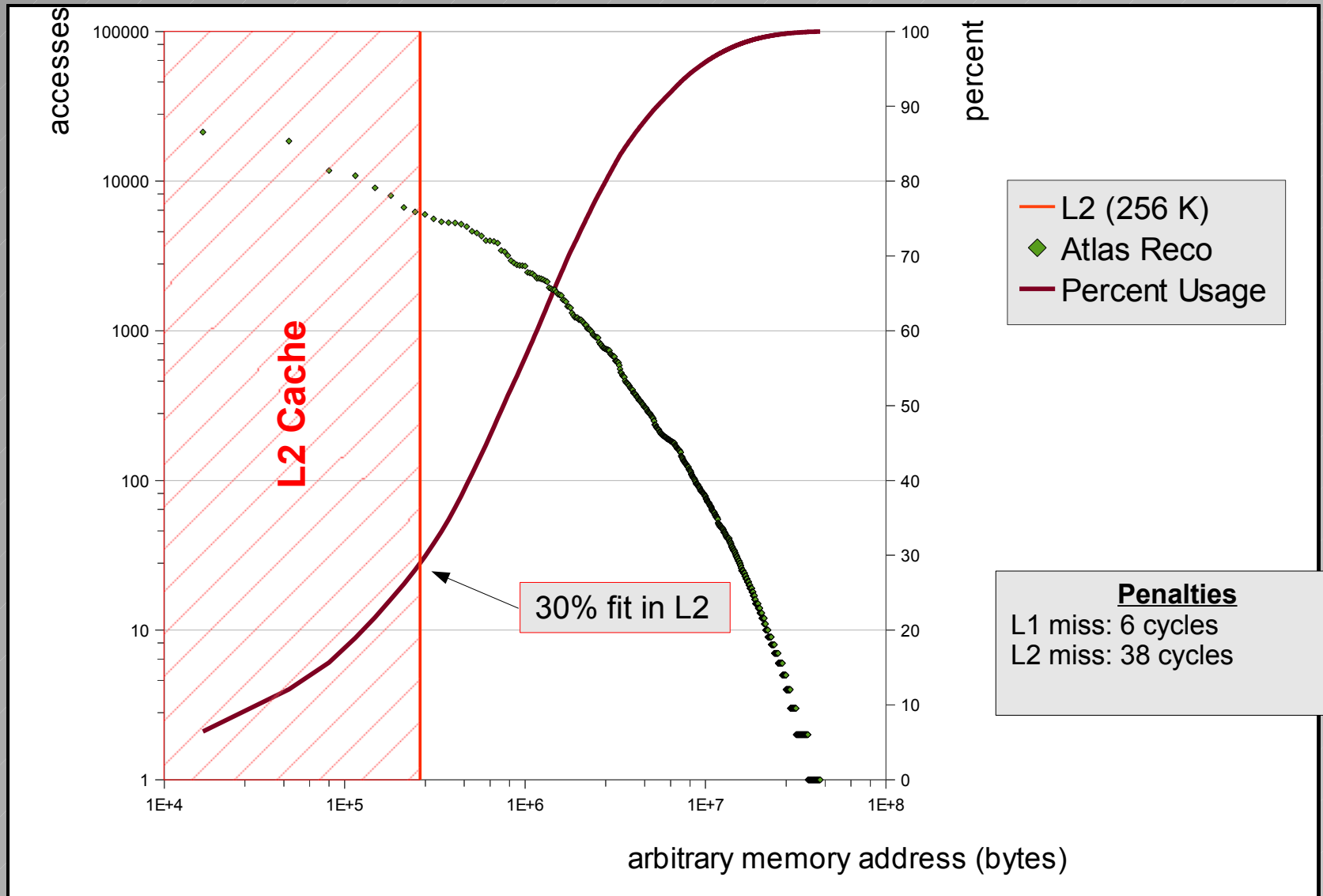
Size of Atlas Reconstruction



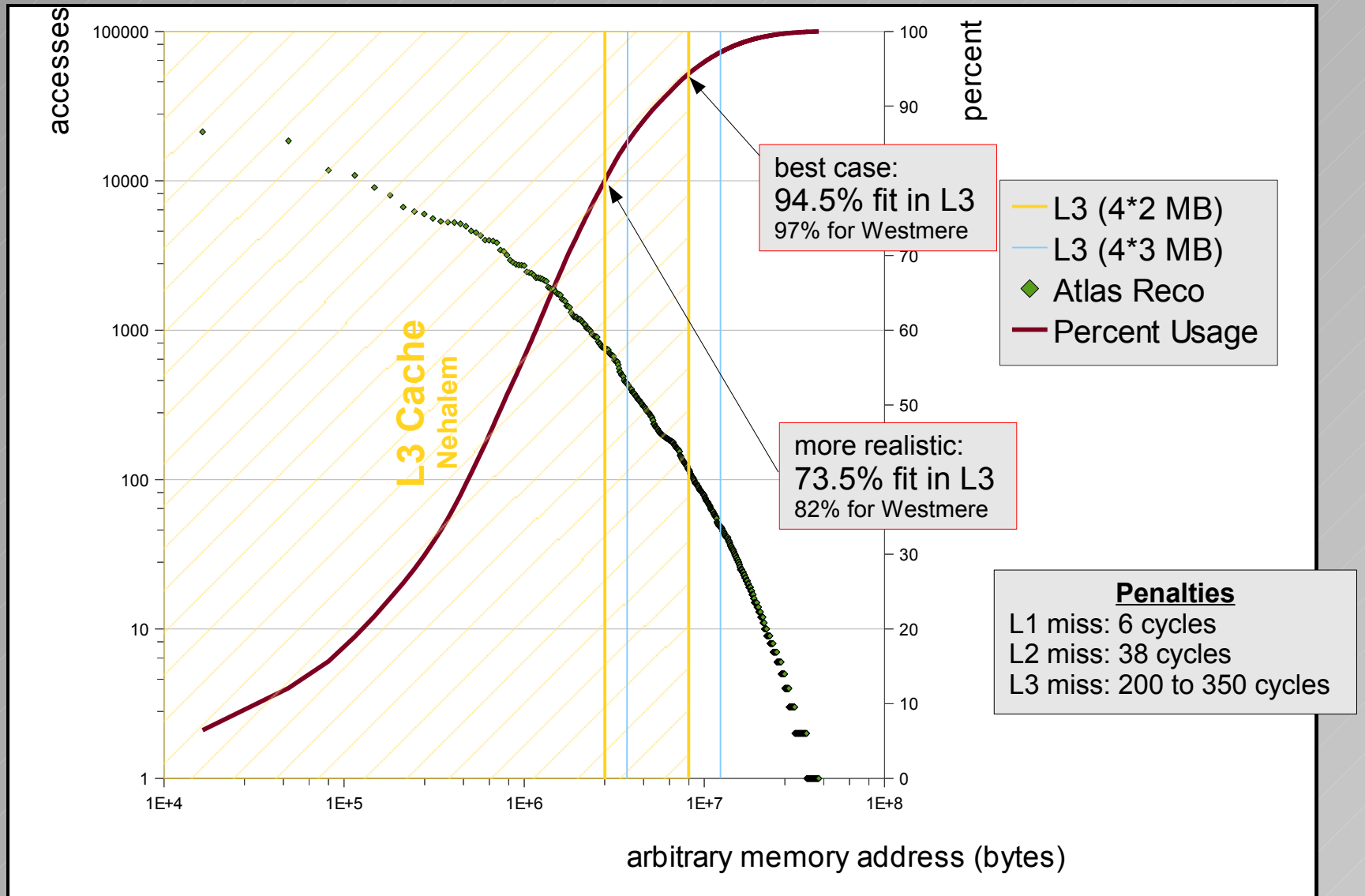
Size of Atlas Reconstruction



Size of Atlas Reconstruction



Size of Atlas Reconstruction



Optimizing Large Object Oriented Code



- Inlining used to be the advice of choice but things are more complicated
 - Even if fewer in overall number
- Inlining increases binary size and can make ifetch misses more costly and code slows down
 - Even if fewer in overall number
- Large codes built of many small methods can result in flat cycle profiles
 - It can take thousands of functions to account for 80% of the clock cycle samples
 - Thus thousands of functions must be optimized to achieve a significant performance improvement

Issues with Large OOP Code Bases



- Function calls result in added instructions
 - Call and return
 - Runtime address resolution (trampolines) required for position independent code/ shared object cross invocations
 - Indirect branches can be more costly
 - Freeing & restoring registers for local use
 - Setting and reading function arguments
- Virtual function calls (function pointers) increase indirect call instructions and associated pointer loads
 - Virtual functions can't be inlined!
- Atlas code has 2500 shared libraries!

Observations from PTU

- In Atlas code, functions are on average only 33 instructions long
- Overhead for function calls is anywhere between 6 and 12 instructions
 - We can have up to 35% overhead!
- We also see instruction starvation of about 20%

Detecting OOP Inefficiencies

- Classic OOP will result in code bases of small functions integrated together to invoke the algorithm
- With the help of experts at Intel we have developed a series of signatures that identify these inefficiencies using hardware counters and PTU

```
Low instruction_retired / call_retired  
High call_retired / branch_retired  
High indirect_call / call_retired  
High uops_issued.core_stall_cycles - resource_stalls.any  
    measures instruction starvation in pipeline  
High  $\sum \text{latency}(\text{source}) * \text{ifetch\_miss}(\text{source})$ 
```

- All these are present in Atlas code (as well as other LHC codes)

Conclusions: Short Term Solutions

- Use social network analysis/network theory to identify clusters of active, costly function call activity
- Order clusters by total time and/or total “cost”
 - Split time of functions shared between clusters by call counts
 - Calls have a direction
 - Utility functions must not be viewed as bridges
- Manually reduce function count in hot clusters by explicit code inlining
- Prioritize work by call overhead cost to be gained
- Duplicate code as needed
- Reduce cross shared object call counts

Conclusions: Longer Term Solutions

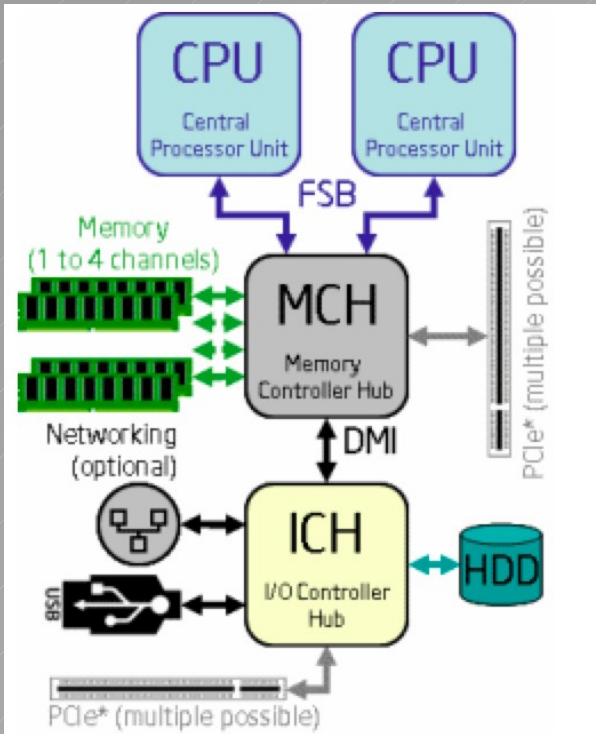


- We (HEP) are not the only ones facing these problems
 - Oracle, IBM, Google all have similar issues
 - They're only just now beginning to realize it
- We need new tools and analysis techniques
 - Current tools fail to show where the problem are, or are not suited to large scale deployment
- We need to drive these optimization techniques into the compilers and linkers themselves
- Changes at the hardware level would also improve the situation
 - It's already happening: new counters are being included in Intel's Westmere and Sandybridge chips which make profiling more useful
 - If we can show Intel exactly what's wrong, and what it will take to fix it in hardware, **they will listen.**

Extra

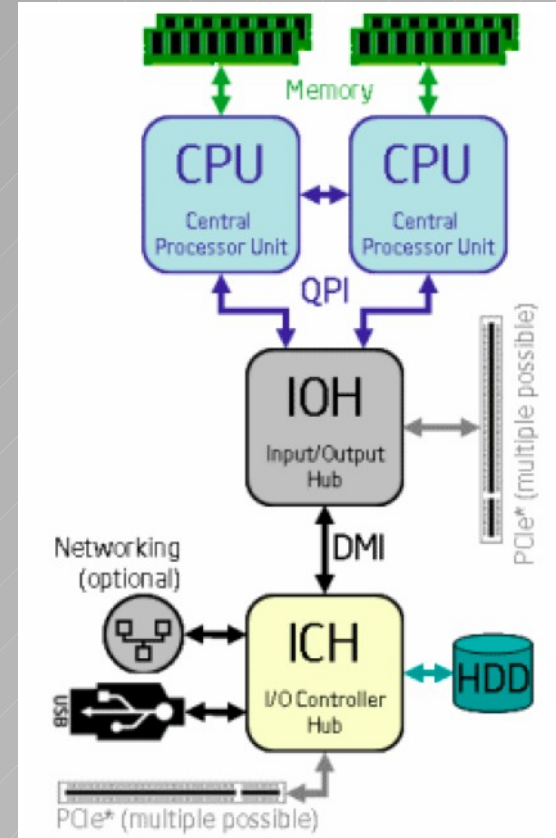
Architecture Upgrades

Intel Cloverfield most of LXPLUS machines



CPU-Memory symmetric access

Intel Nehalem

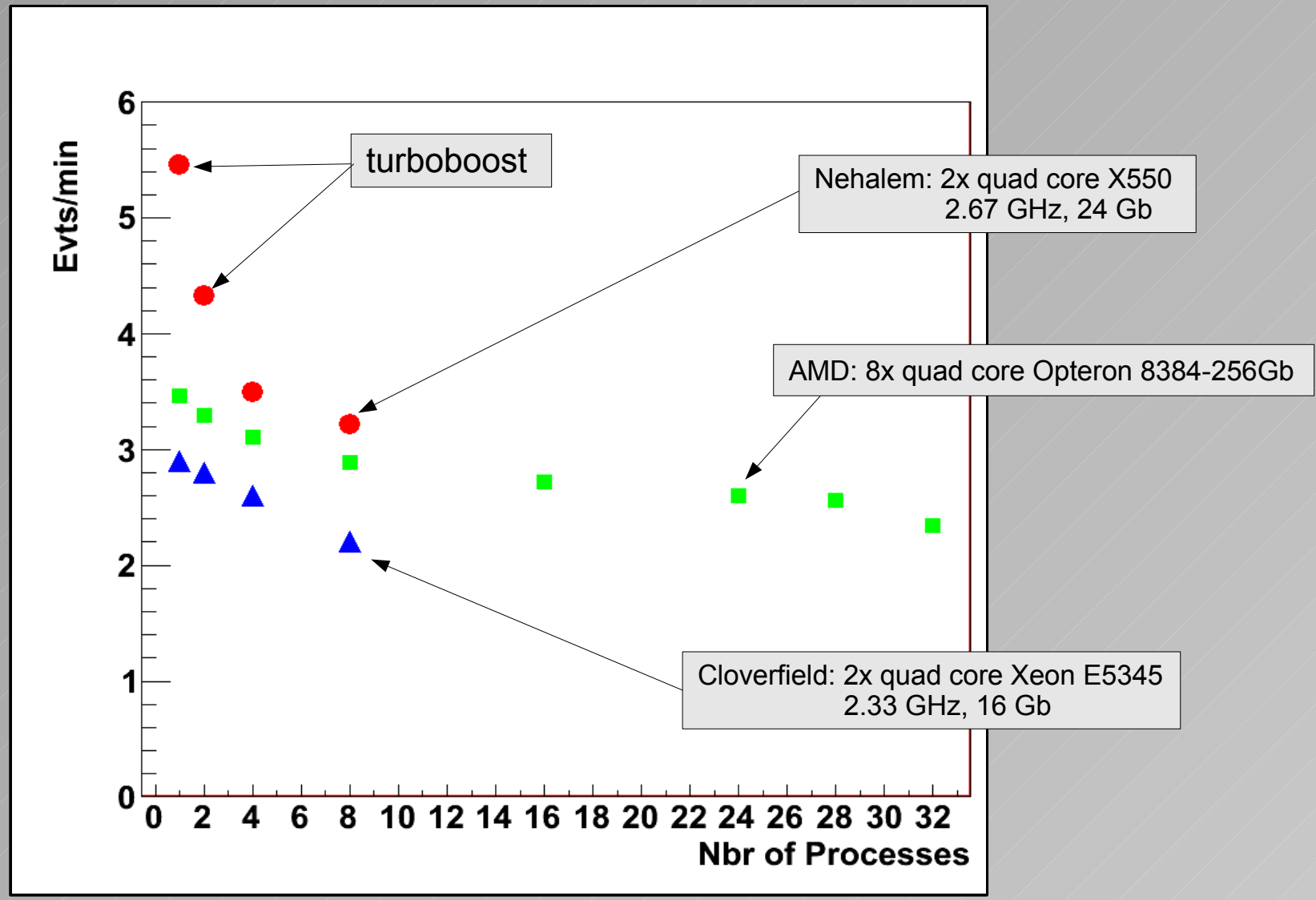


- Hyper Threading -> two logical cores on physical one
- QPI Quick Path from CPU to CPU and CPU-to-Memory
- Turbo Boost -> dynamic change of CPU-frequency
- CPU-Memory non-symmetric access (NUMA)

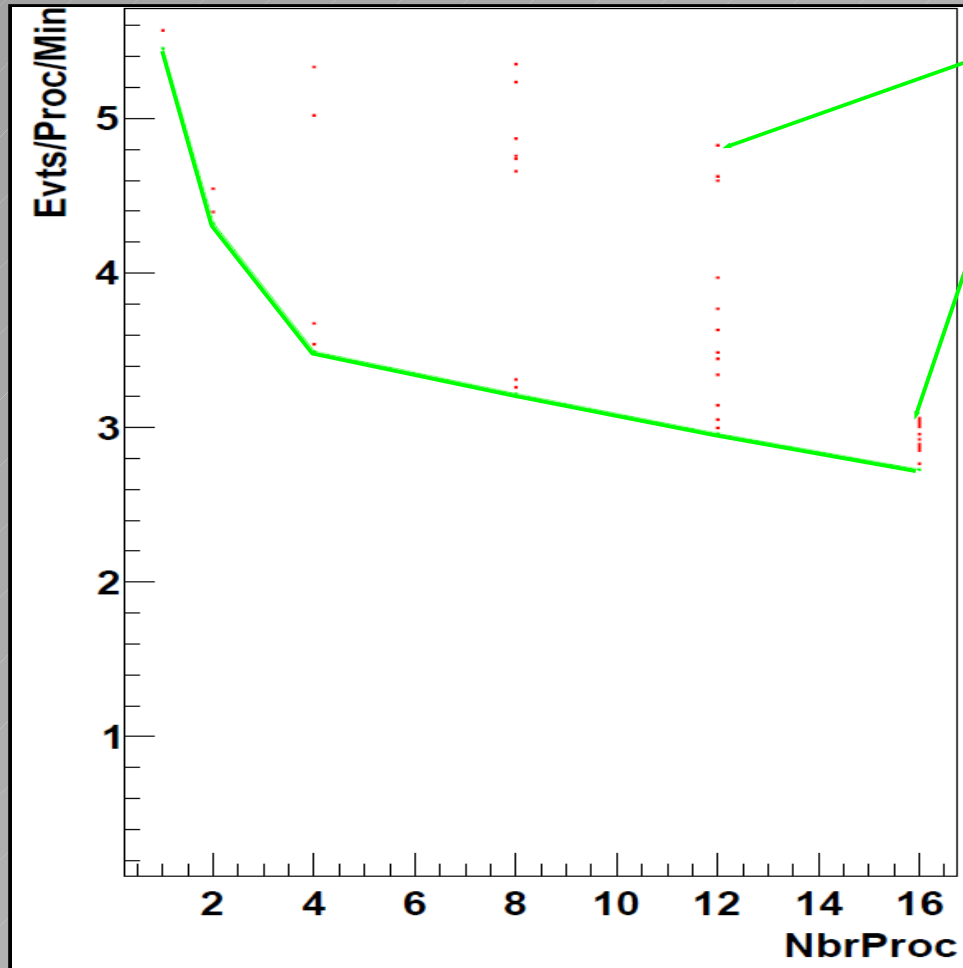
Sub-Event Parallelism

- Event based parallelism requires heavy I/O at end of jobs for merging output files
 - develop parallel I/O mechanisms
- Use separate worker threads to process distinct regions of interest
 - calorimeter, muon, silicon, etc
 - single worker thread to distribute data objects to clients
- Will require significant rewriting of framework code

Event Throughput Per Process for RDO to ESD Reconstruction on Different Machines



Worker Throughput, No Event Queue



individual worker event rates

- 8 core HT machine

Event Distribution Using Queue

```
events = multiprocessing.queue(EvtMax+ncpus)
events = [0,1,2,3,4,...,99, None,None,None,None]
```

...

```
evt_loop(evt=events.get(); evt != None):
    evt_loop_mgr.seek (evt_nbr)
    evt_loop_mgr.nextEvent ()
```

core-0

WORKER 0:
Events: [0, 4, 5,...]

core-1

WORKER 1:
Events: [1, 6, 9,...]

core-2

WORKER 2:
Events: [2, 8, 10,...]

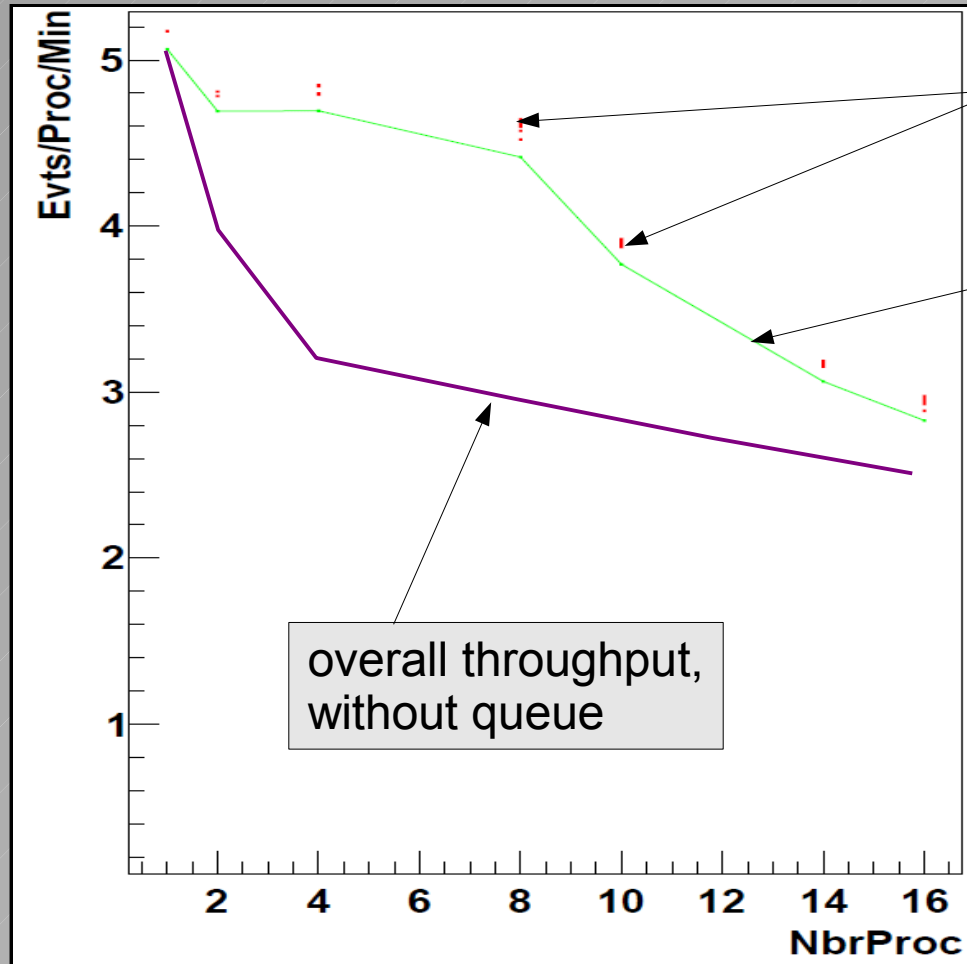
core-3

WORKER 3:
Events: [3, 7, 11,...]

Balance the arrival times of workers!

Slower worker doesn't get left behind

Worker Throughput with Event Queue



individual worker event rates

overall throughput, with queue

overall throughput, without queue

- 8 core HT machine

Intel PTU in Action

Applications Places System 4:51 PM

Intel(R) Performance Tuning Utility - /home/levinth/workspace_4_nda/mlrc_orig/Loop-Analysis-with-Call-Sites-2010-04-29-16-14-42 - Eclipse Platform

File Edit Navigate Project Run Window Help

Loop-Analysis-with-Call-Sites-2010-04-29-16-14-42

Function	RVA	Module	CPU...	CPU...	INST...	UOPS...	UOPS...	UOPS...	MEM...	MEM...	RES...	BR_INST_RETIRED_NEAR_CALL	UOP...	RE...
compute_gen_staple	0x376A	su3_rmd	33,410	33,410	35,287	12,179	20,637	19,907	22,025	22,091	18,632		0	38,163
path_product	0x56BE	su3_rmd	27,360	27,360	30,277	10,763	16,813	17,079	22,579	22,604	14,609		1	31,494
u_shift_hw_fermion_pp	0x15150	su3_rmd	21,156	21,156	26,444	9,948	11,882	11,709	16,040	16,107	11,133		6	27,959
eo_fermion_force_3f	0x13972	su3_rmd	0	0	0	0	0	0	0	0	0		3	0
eo_fermion_force_3f	0x138E7	su3_rmd	0	0	0	0	0	0	0	0	0		1	0
eo_fermion_force_3f	0x137F3	su3_rmd	0	0	0	0	0	0	0	0	0		2	0
dslash_fn_on_temp_s...	0xC044	su3_rmd	8,870	8,870	20,017	733	2,167	2,217	592	583	1,873		1	22,164
add_3f_force_to_mo...	0x14842	su3_rmd	16,839	16,839	28,255	3,984	6,240	5,866	4,806	4,775	1,837		6	36,652
u_shift_hw_fermion_np	0x16A4E	su3_rmd	7,253	7,253	9,046	3,136	3,882	3,915	5,249	5,232	3,688		5	9,621
imp_gauge_force	0x11AC8	su3_rmd	3,621	3,621	3,539	1,418	2,171	2,223	1,843	1,820	1,752		0	5,067
eo_fermion_force_3f	0x12768	su3_rmd	3,543	3,543	5,576	355	1,017	1,097	407	374	783		0	8,081
<unknown(s)>	0x0	vmlinux	4,613	2,268	2,136	599,612	458,805	731,810	1,102	713	416		85,098	4,003
add_3f_force_to_mo...	0x16144	su3_rmd	6,414	6,414	11,425	1,269	2,158	2,077	1,462	1,476	808		2	14,722
add_3f_force_to_mo...	0x170EE	su3_rmd	4,441	4,441	8,076	822	1,403	1,337	951	932	450		0	10,444
declare_strided_gather	0x73F4	su3_rmd	783	783	1,815	198	167	125	30	32	115		48	1,791
load_longlinks	0x5150	su3_rmd	410	410	262	224	289	296	348	349	214		0	375
add_3f_force_to_mo...	0x157F2	su3_rmd	1,434	1,434	2,549	278	452	459	313	315	122		0	3,294
dslash_fn	0x8388	su3_rmd	470	470	576	158	266	268	185	186	237		0	629
grsource_imp	0xED88	su3_rmd	260	260	123	134	219	208	251	249	181		0	152
update	0xA40A	su3_rmd	156	156	97	85	119	109	134	134	99		0	144

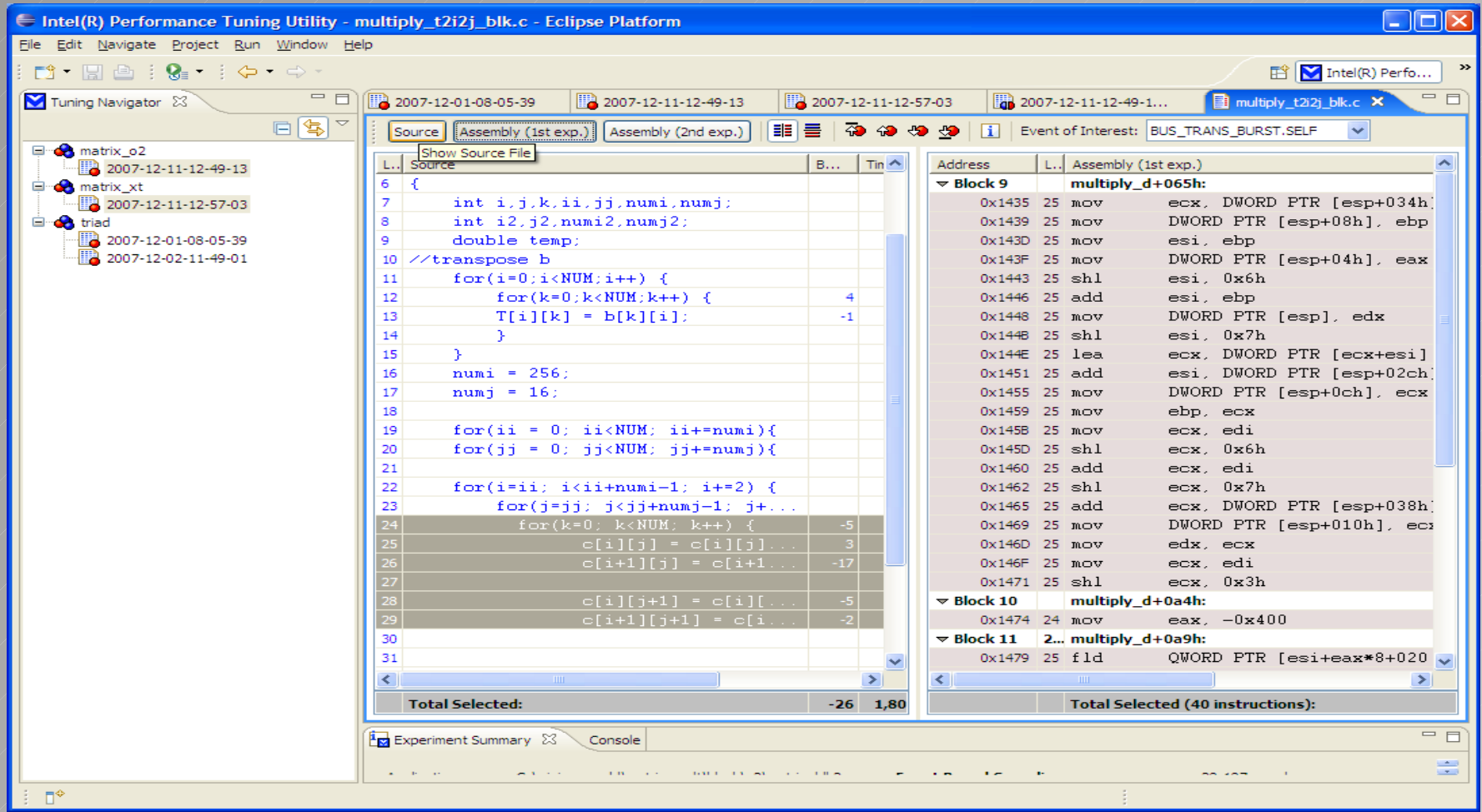
Limit 95% Granularity Function Process All Thread All Module All Cpu Total

Experiment Summary Console Advanced Profile Info

```
<terminated> Intel(R) Core(TM) i7 processor family - Loop Analysis with Call Sites [Intel(R) PTU] vtsarun /mlrc_orig/Loop-Analysis-with-Call-Sites-2010-04-29-16-14-42 -s -dl -ec ARITH_CYCLES_DIV_BU
--- workload ---
workLoad stopped => 04/29/2010 04:28:53 PM
```

levinth@levinth-nhmb:~ Intel(R) Performance Tuning Utility - /home/levinth/worksp... Starting Take Screenshot

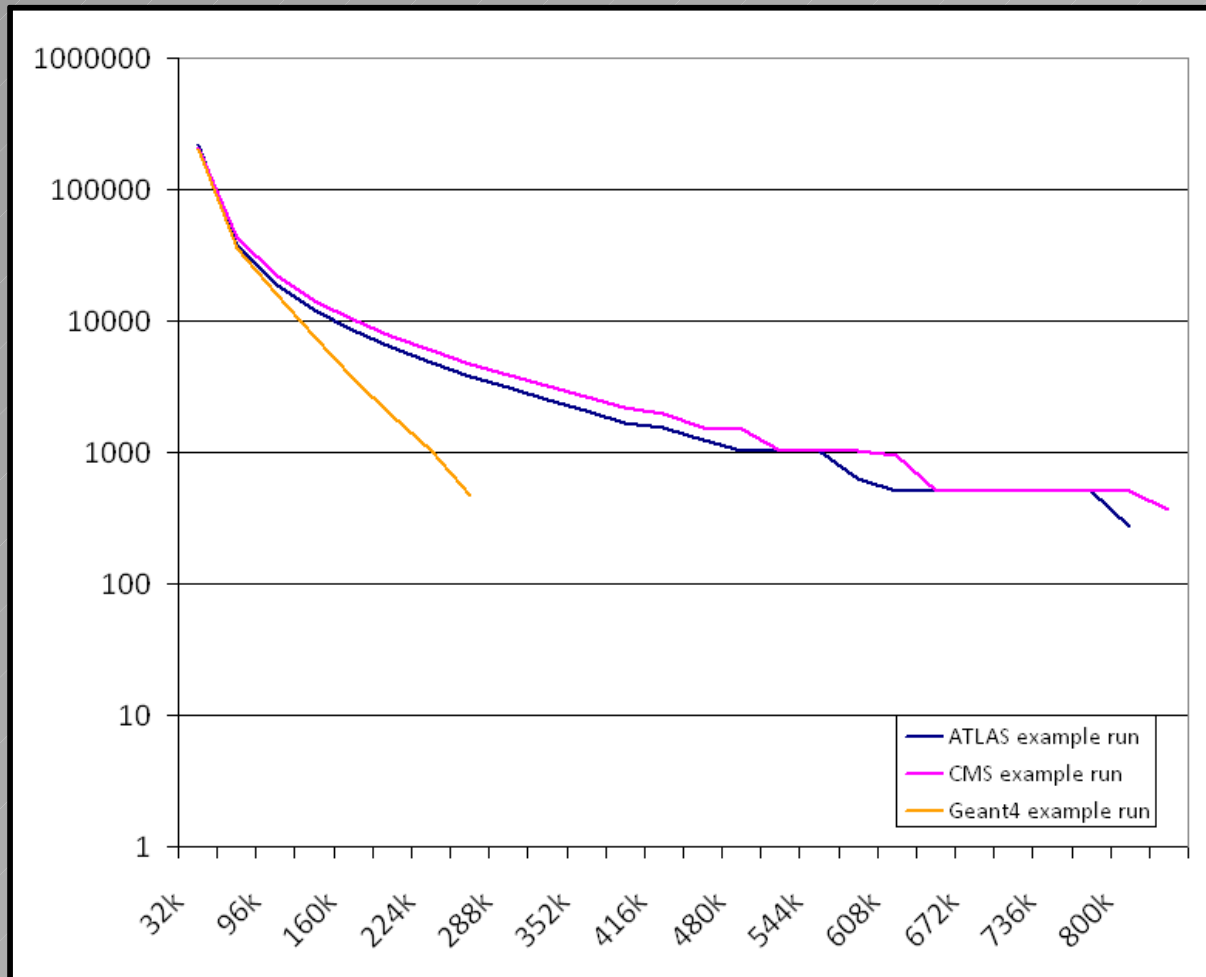
Intel PTU Source View



The screenshot displays the Intel(R) Performance Tuning Utility (PTU) interface within the Eclipse Platform. The main window is titled "multiply_t2i2j_blk.c - Eclipse Platform". The interface is divided into several panes:

- Left Pane (Tuning Navigator):** Shows a project tree with folders like "matrix_o2", "matrix_xt", and "triad", and sub-folders with dates.
- Top Pane (Source View):** Displays the C source code for "multiply_t2i2j_blk.c". The code includes variable declarations for `i, j, k, ii, jj, numi, numj`, a `double temp;` declaration, and nested loops for matrix multiplication. A selection box highlights lines 24-29, with a "Total Selected: -26 1,80" status bar at the bottom.
- Right Pane (Assembly View):** Shows the assembly code for the selected region, titled "multiply_d+065h:". It lists instructions such as `mov ecx, DWORD PTR [esp+034h]`, `mov esi, ebp`, `mov ecx, DWORD PTR [ecx+esi]`, and `fild QWORD PTR [esi+eax*8+020]`. A selection box highlights lines 24-29, with a "Total Selected (40 instructions):" status bar at the bottom.
- Bottom Pane:** Contains "Experiment Summary" and "Console" tabs.

Size of CERN programs



Cacheline access frequency evaluated by sorting cachelines by their accesses
 Thus a binary working set size measurement

Atlas PTU Results Overview

0.8969516205	cycles/instruction	
0.5032359284	retirement_stall_cycles/all_cycles	
instruction starvation stall evaluation		
0.1990232561	Instruction_starvation_stall_cycles/all_cycles	
0.0637655987	L2_ifetch_hit*6/all_cycles	
0.0434461272	llc_ifetch_hit*38/all_cycles	
0.0046586531	ifetch_local_dram*200/all_cycles	
0.0003089745	ifetch_remote_dram*350/all_cycles	
0.0062712638	itlb_miss*35/all_cycles	
	0.1184506174	sum of ifetch penalties/all_cycles
load latency stall evaluation		
0.0182017861	Load_l2_data_hit*6/all_cycles	
0.0542206392	Load_LLC_data_hit*38/all_cycles	
0.0557115481	Load_local_dram_data_hit*200/all_cycles	
0.0001682534	Load_remote_dram_data_hit*350/all_cycles	
	0.1283022268	sum of load data fetch penalties/all_cycles
6.4360070187	instructions/branch	
33.3007279867	instructions/call	
5.1741286002	branches/call	
0.1596675103	fraction of indirect calls	
0.116038481	fraction of indirect branches	
0.0213900884	fraction of mispredicted branches	