ORACLE

An Experimental Model to Analyze OpenMP Applications for System Utilization

Mark Woodyard
Principal Software Engineer

The following is an overview of a research project.

It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

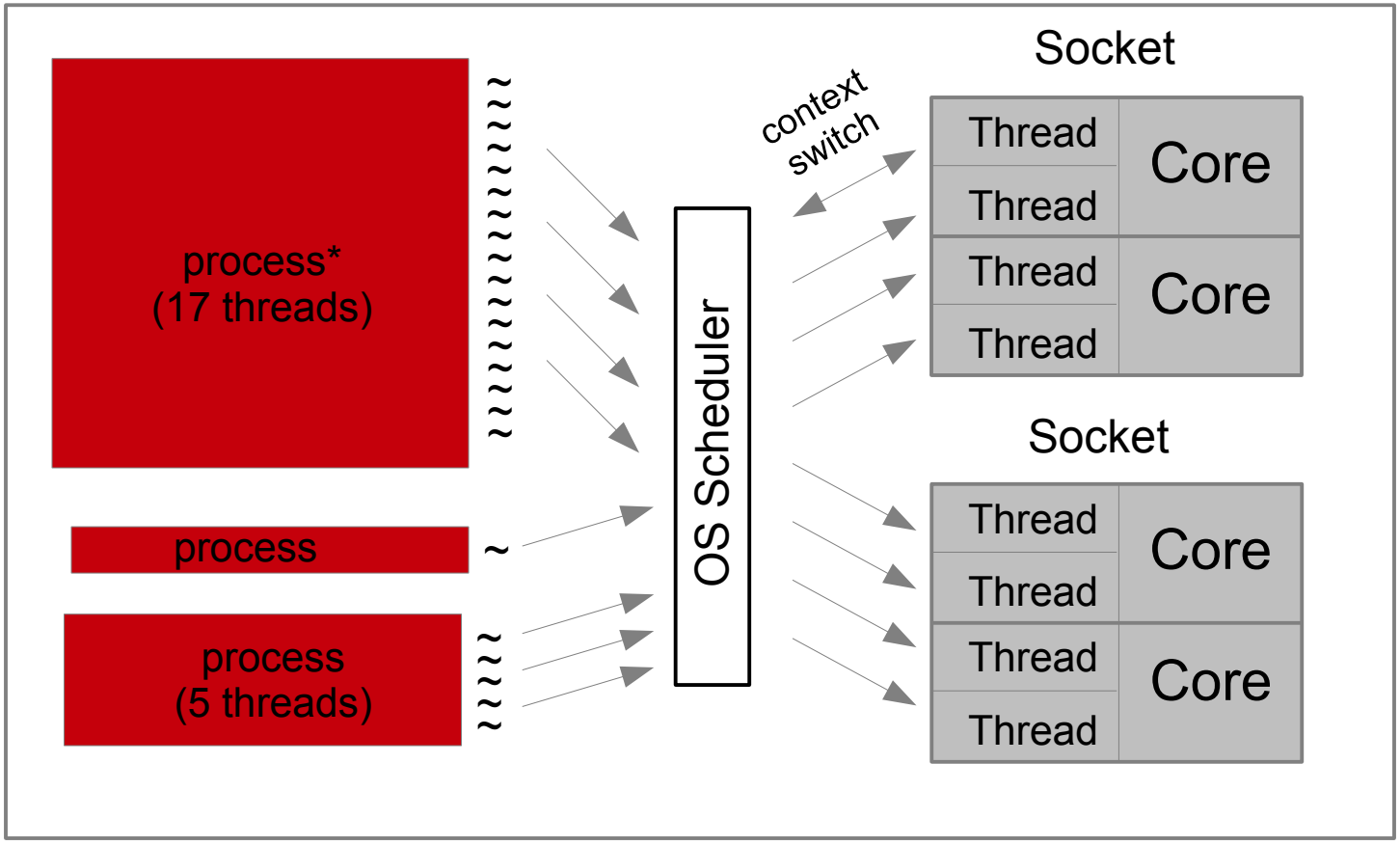
The opinions expressed here are my own and do not express the views of Oracle Corporation.

The Importance of System Utilization

- No program is 100% parallel
  - Utilization unimportant for one small, cheap server...
  - Critically important for a large data center
- No OS can provide 100% cycles to applications
  - Focus on single application performance
  - Lean on processor sets, processor binding
  - Result: reduced overall utilization
- Core counts per socket are increasing
- Systems are getting larger: more sockets & cores
- More than many applications can efficiently use

OpenMP Performance in a Throughput Context

Operating System Scheduler: Software Threads → Hardware Threads
Scheduling & Work Distribution Interactions Impact Utilization



*Process = Executing Program ~ = Executing Thread Process Memory

Need: Enable High Parallel Utilization

- Philosophy: Make efficient use of *any available* cycles
- If my program doesn't, how do I find the cause?
- Key scalability questions:
  - How well does the entire program scale?
  - Which parallel region is a bottleneck?
  - Do I have enough parallel work vs. overhead?
  - Is the serial time significant vs. parallel work?
  - Are the work chunks large enough vs. the time quantum?

Comparison of Scaling Analysis Models

Dedicated Analysis

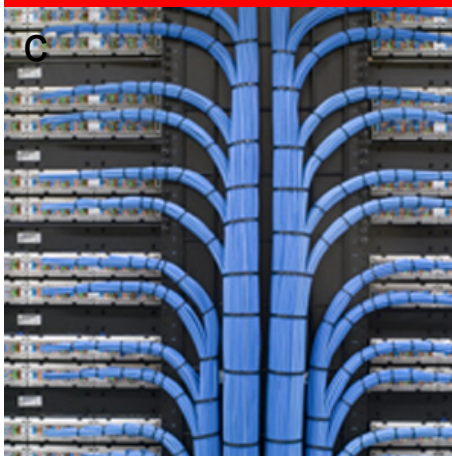
- Time to Solution
- Run at 1, 2, … N threads
  - One thread per core
  - Compare speedup at each thread count
- Scaling Reference: traditional 1 vs. N cores
- Scaling Bottleneck: Ambiguous: OpenMP or hardware

Non-Dedicated* Analysis

- Usage of Available Cycles
- Run only at N threads
  - Up to N cores
  - Model speedup at <= N cores
- Scaling Reference: loaded system
- Scaling Bottleneck: Unambiguous: OpenMP only

*Non-Dedicated is called Throughput

Tool Overview



Experimental Tool

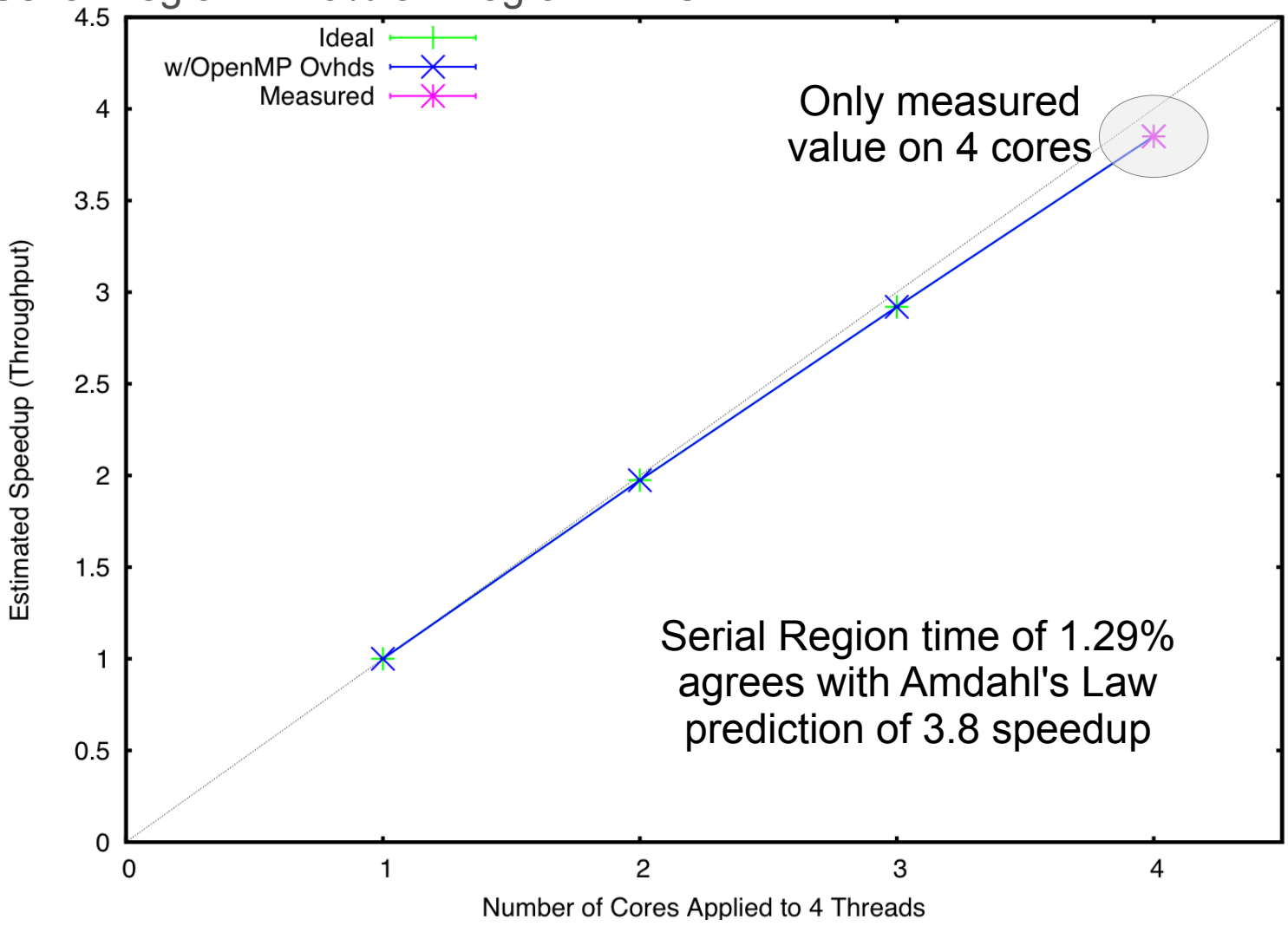
- Experimental tool implements the model
  - Runs on Solaris (x64 & SPARC)
- Model applies to any UNIX system
- Tool generates shell/DTrace script to run target program
- Trace data processed to generate performance model:
  - Reconstruct the OpenMP program structure as run
  - Simulate execution on a range of core counts (round-robin)
  - Generate a text report, and optionally sequence of graphs
- Speedup analysis aggregated to:
  - Each worksharing construct
  - Each parallel region
  - Entire program
- Complimentary to existing tools

Terms Used: What is Speedup?

- Speedup: ratio of *work time* to *elapsed time*
- Work Time:
  - Any time not OpenMP overhead
  - Measured as run with N threads
- *Ideal* Speedup: without OpenMP overhead
- Elapsed Time:
  - Measured (as run), presuming all cores available
  - Estimated via simulation (1 thru N cores using times as run)
- *Preceding Serial*:
  single-thread time before a parallel region
- *Stand-Alone*:
  loop speedup without considering preceding serial time

Example Tool – Speedup Analysis Entire Program

Serial Region: 1.29% of Program Time



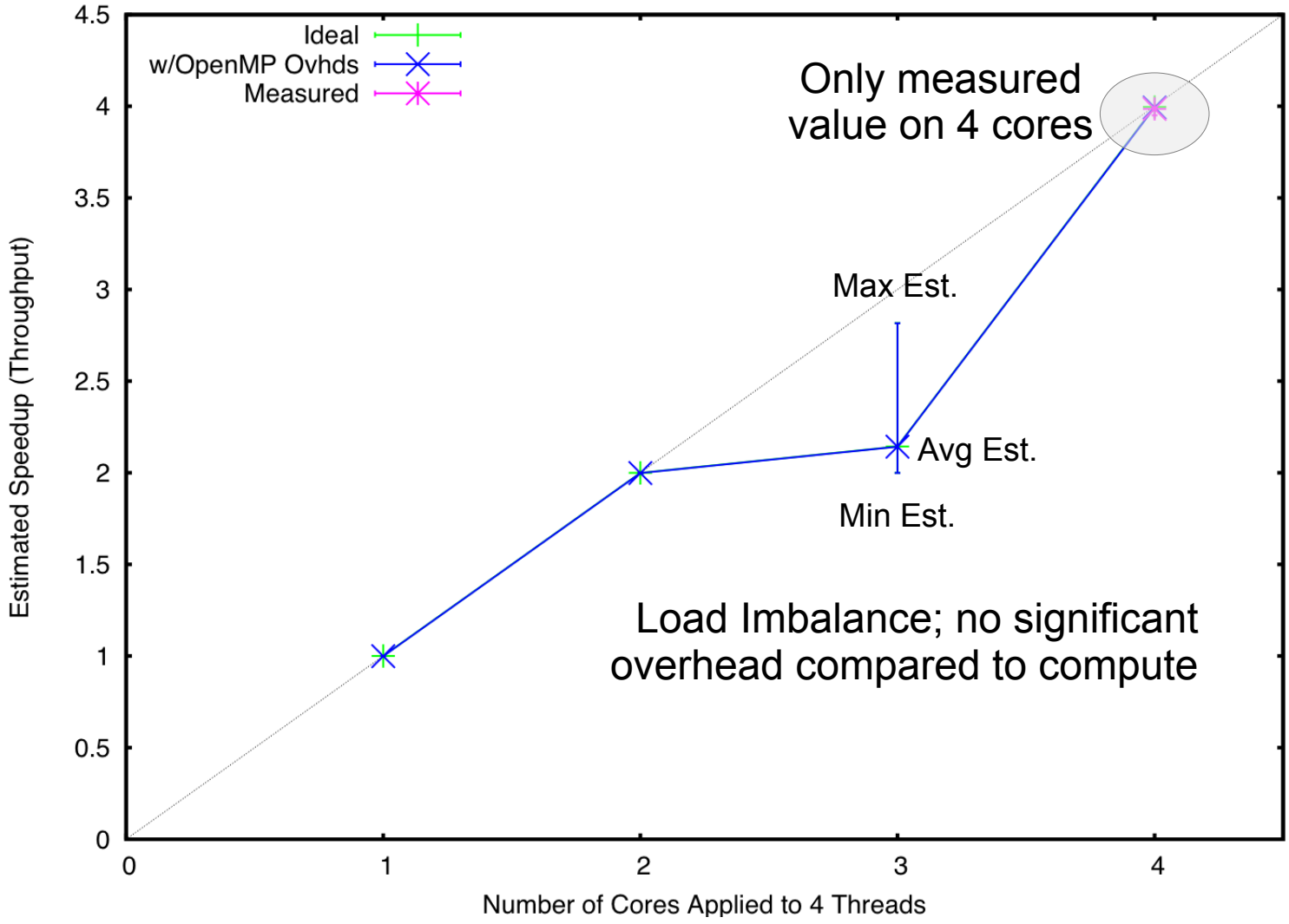
About the Charts

Maximum value measured or estimated

Average value measured or estimated

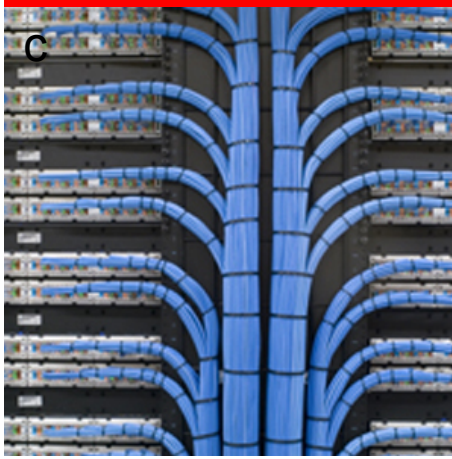
Minimum value measured or estimated

Example Tool – Detail for a Work Share Loop with Load Imbalance



How To Measure Utilization on Solaris

- Use `cputimes` script from DTrace Toolkit
- http://opensolaris.org → Community Group → DTrace
- Measures all cycles for duration of script:
  - User threads
  - Daemons
  - Kernel
  - Idle time
- Utilization: Percent of non-idle cycles during run
- Sanity check: daemons & kernel are small fraction
- Measurement system: Sun Ultra 27
  - Intel quad-core Xeon W3570 processor (3.2 GHz)
  - Processor threading turned off (one thread per core)
  - Solaris 10



Example Program:
Time Share Scheduling Class

Example Program: w/Parallelized Fill Loop, n=5000

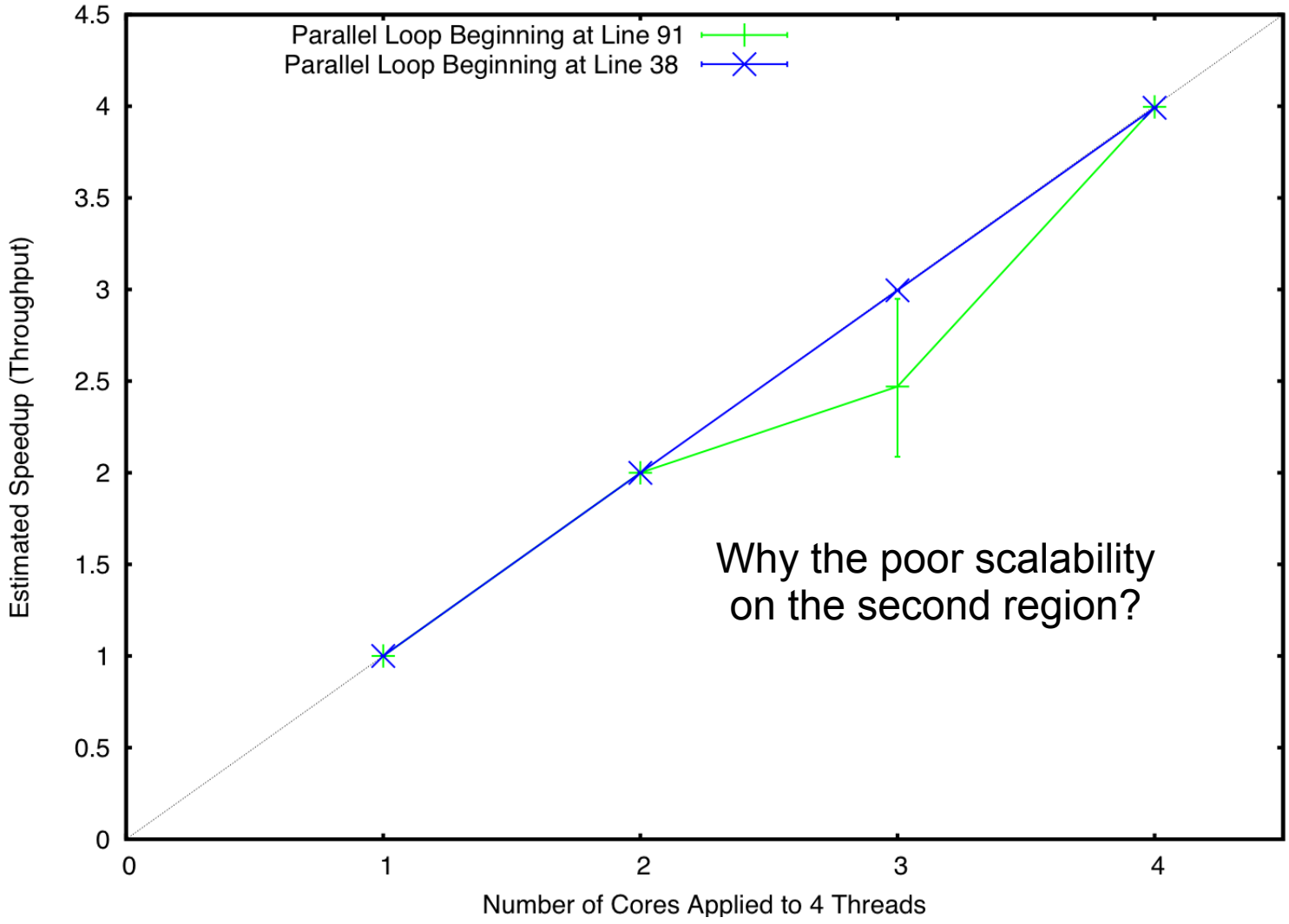
```
22 !$omp parallel shared(a,b,c,n,p) private(i,j,k,z)
...
38 !$omp do schedule(runtime)
39    do k = 1, n
40       do i = 1, n
41          do j = 1, n
42             c(j,k) = c(j,k) + a(j,i) * b(i,k) * z + p
43          end do
44       end do
45    end do
46 !$omp end do
...
71 !$omp end parallel
```

Both regions should scale very well

```
91 !$omp parallel do shared(a,pi,n) private(j,i) schedule(runtime)
92   do j = 1, n
93      do i = 1, n
94         a(i,j) = sqrt(2.0d0/(n+1))*sin(i*j*pi/(n+1))
95      end do
96   end do
97 !$omp end parallel do
```

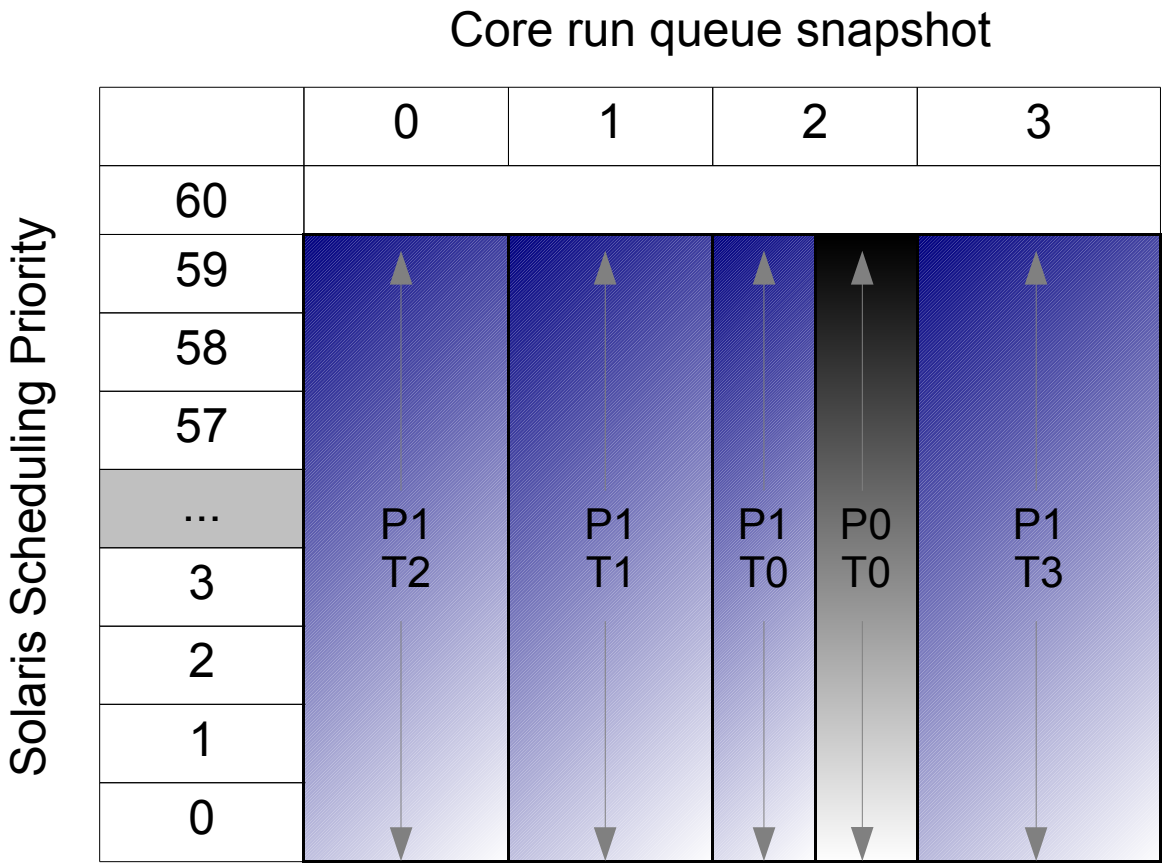
Estimated Speedups: Static Work Distribution

Parallel Loops of Lines 38 and 91



Four Thread & Single Thread Processes

Time-Share Scheduler View

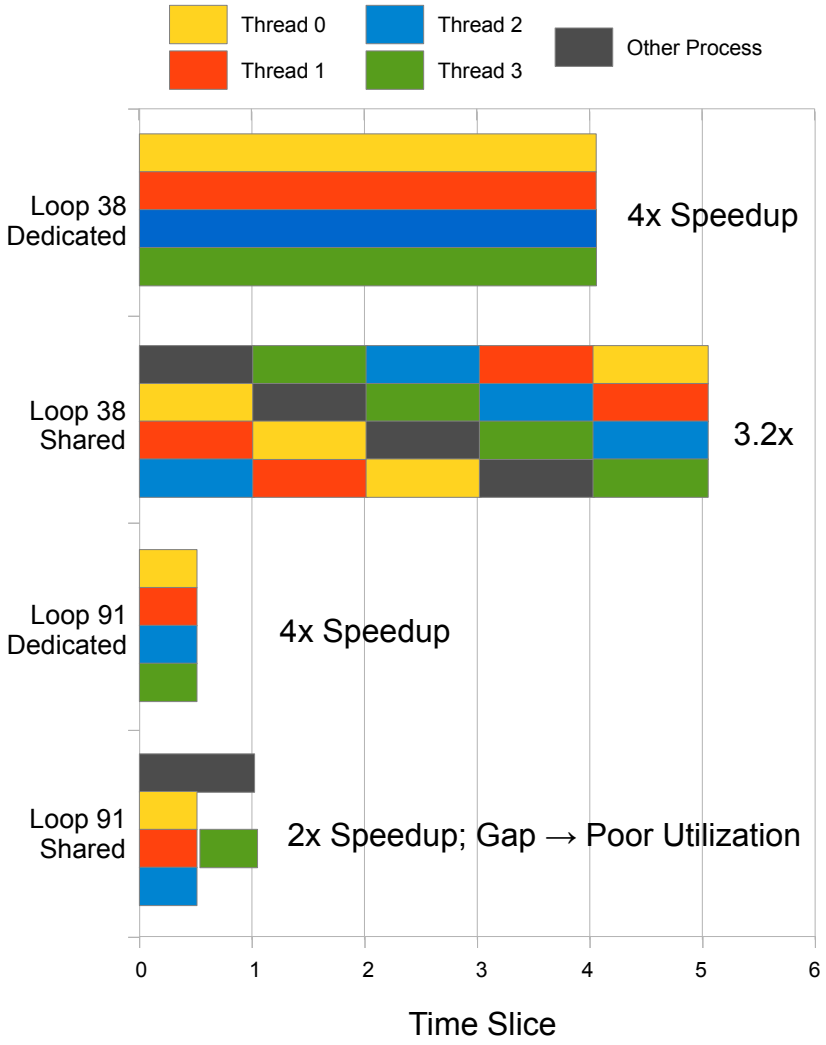


- Process 0: single threaded (infinite loop)
- Process 1: 4-threads static schedule
- Two processes share one core
- Why the poor scalability on the second loop?

OpenMP Static Work Distribution & TS Scheduler

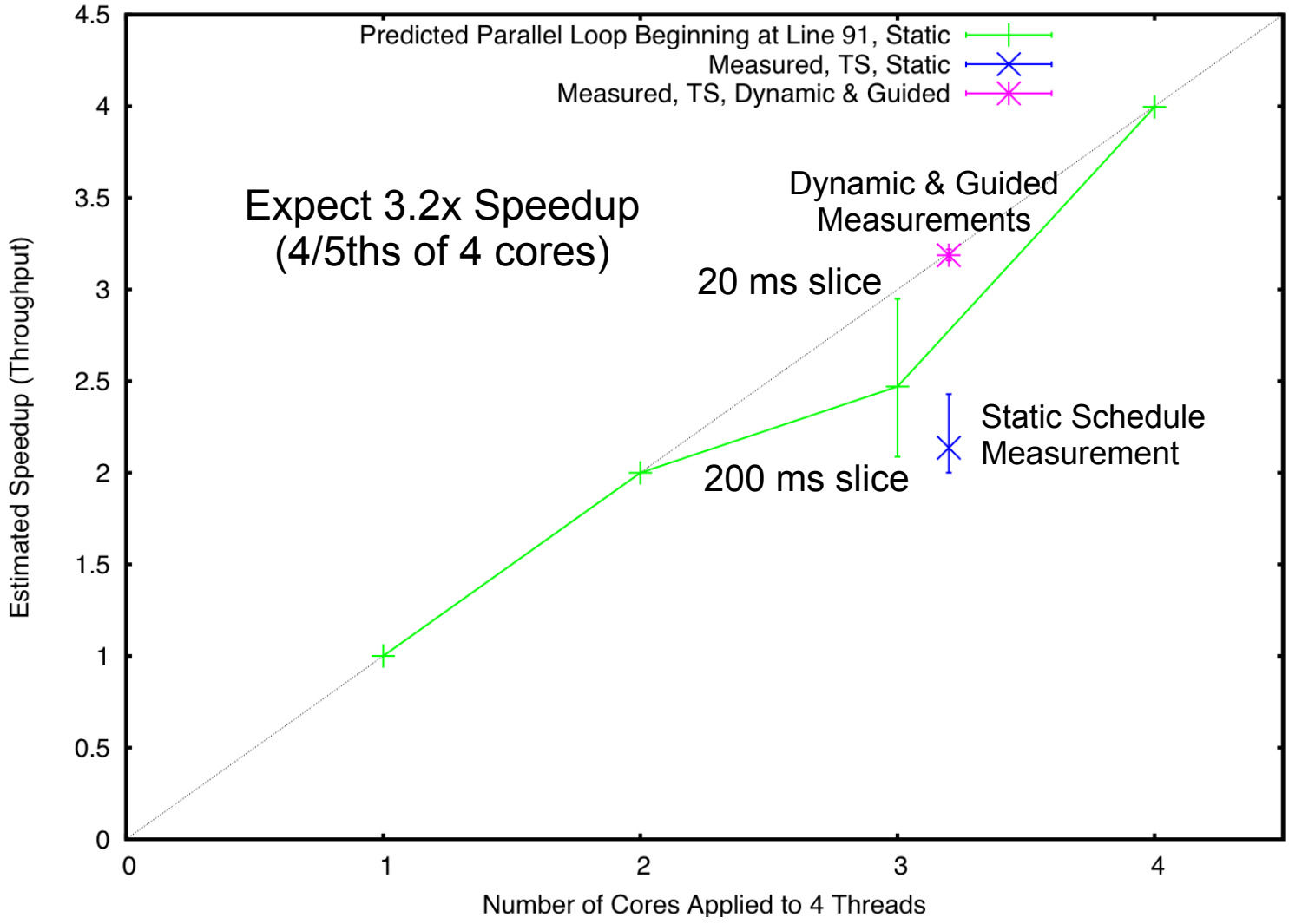
Four Threads Sharing Four Cores With One Other Process, Static

- Solaris time slices:
  - 20 millisec for high priority
  - 200 millisec for low priority
- Loop 38 work: >16 sec
- Loop 91 work: ~57 msec
- Time Share forces compute-intensive threads to low priority
- Result: load imbalance for work chunks < time slice
- Does this really happen?



TS Experiment: Fill Loop of Line 91

Isolated Loop Run with 4 Threads along with Single-Threaded Process

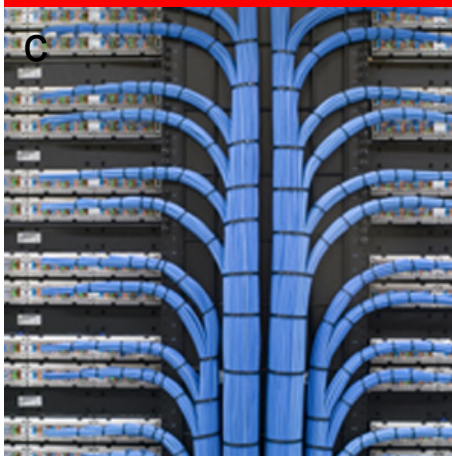


First Insight

Any statically scheduled work-sharing construct can suffer from load imbalance if the time to execute each thread's work is small compared to the time quantum.

How would a programmer identify this situation without measurement?

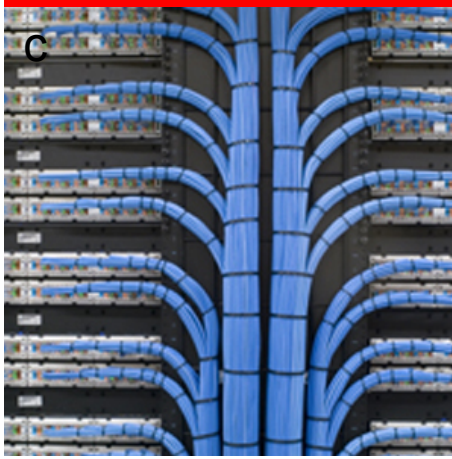
What other problematic scenarios occur?



Time Share
System Utilization Results

TS System Utilization: Isolated Loop 91

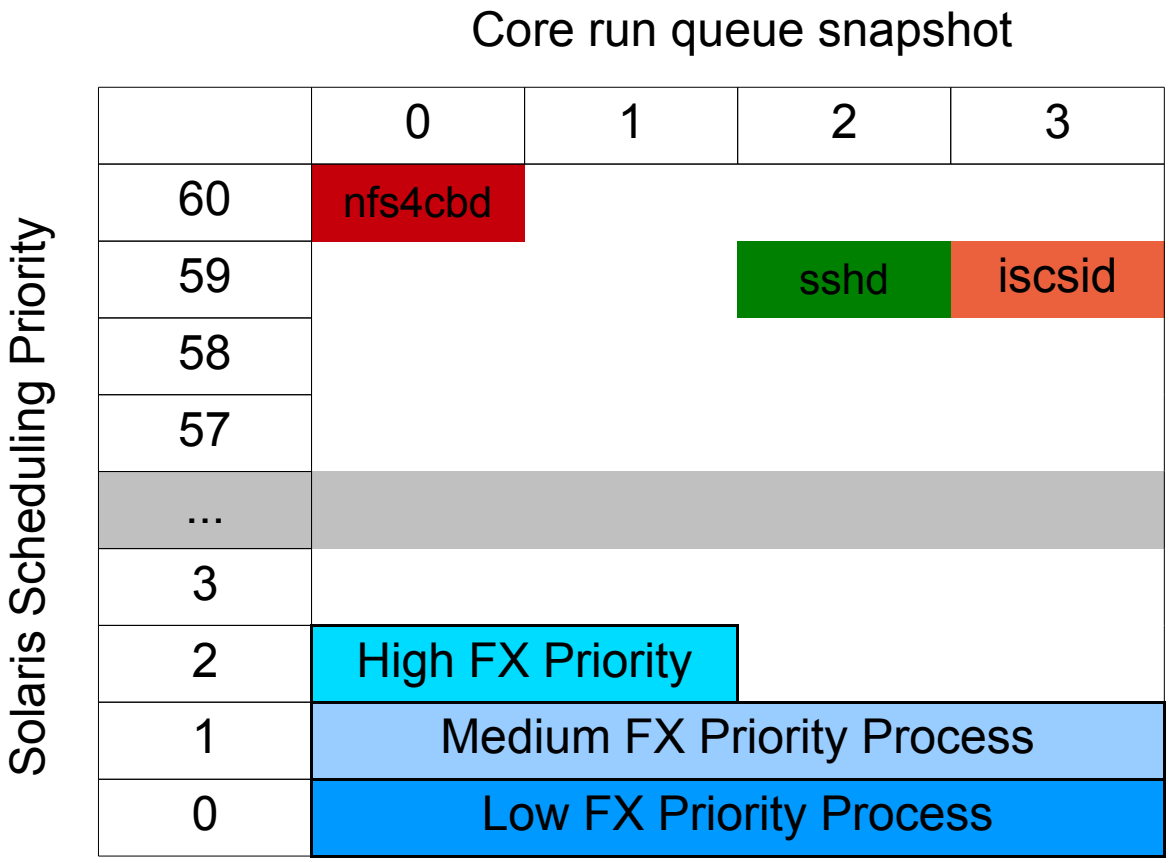
- Utilization measurement over 60 seconds
- Dynamic & Guided utilization: 99.9%
- Static allocation utilization measurement: 86%
- That's a 14% loss in utilization!
- Measured overhead:
  - Dtrace command: 0.0038% processor cycles
  - Kernel & daemons: 0.04%



Example Program:
Fixed Priority Scheduling Class

Solaris Fixed-Priority (FX) Scheduling Class

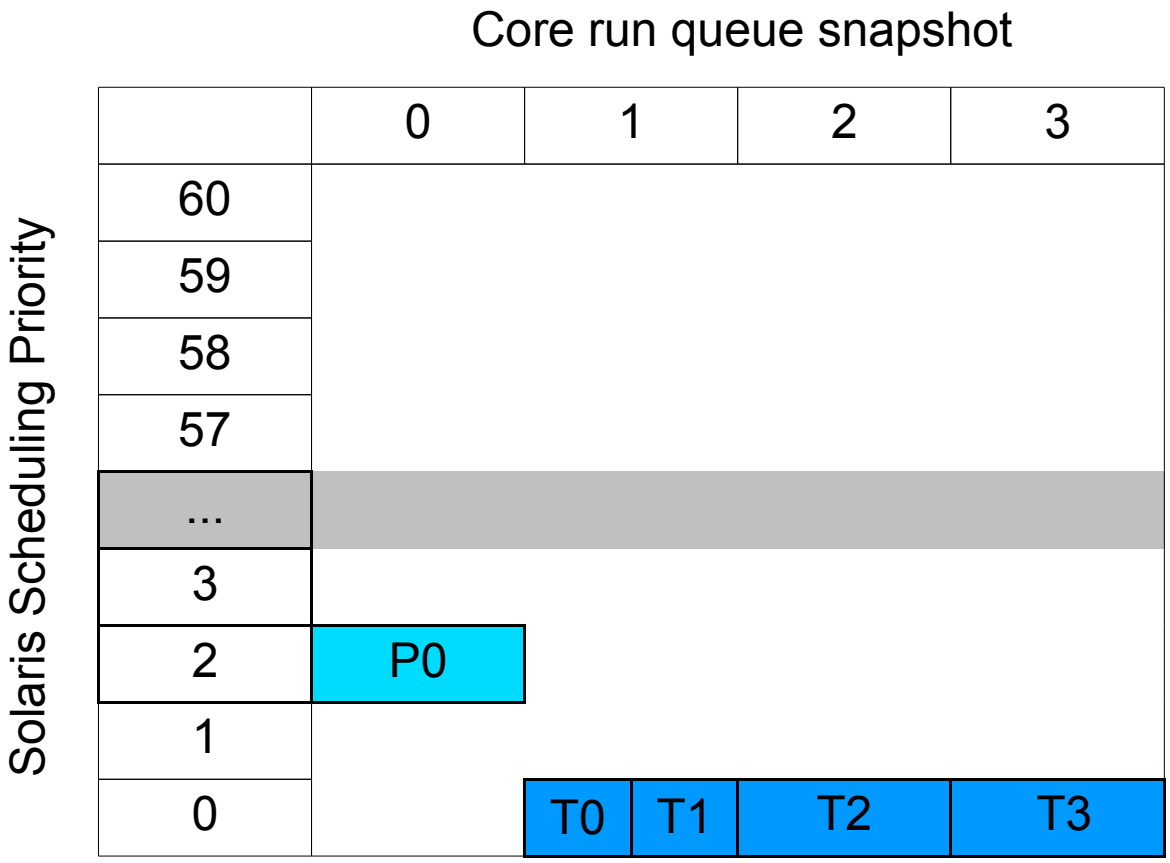
Scheduler Never Adjusts Thread Priority



- Daemons run in TS, as needed
- Highest FX threads run "dedicated"
- Lower-priority threads get unused cycles
- Example:
  Priority 2: 2 thr
  Priority 0,1: 4 thr

Four Thread & Single Thread Processes

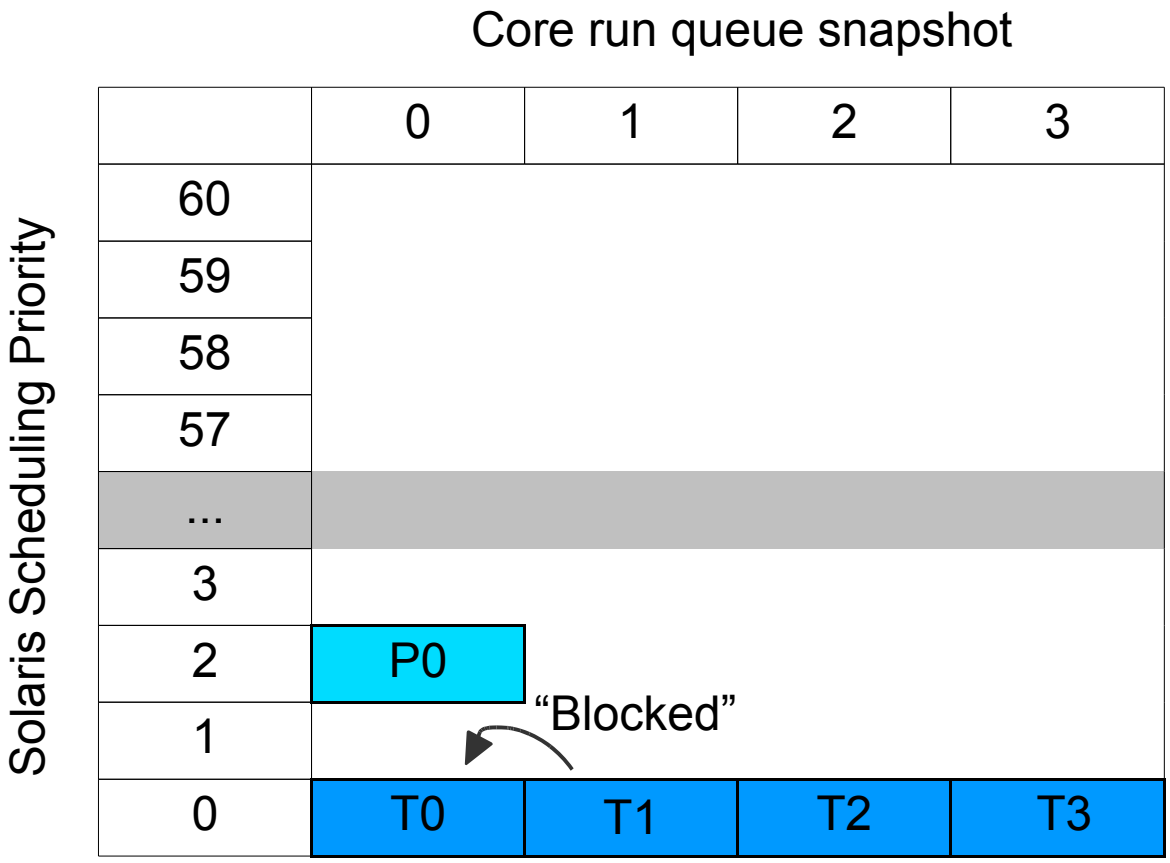
Fixed-Priority Class Scheduler View



- Process P0: Fixed Priority 2 (infinite loop)
- Process P1: Four threads Fixed Priority 0
- Two P1 threads share one core

Four Thread & Single Thread Processes

Fixed-Priority Class Scheduler View

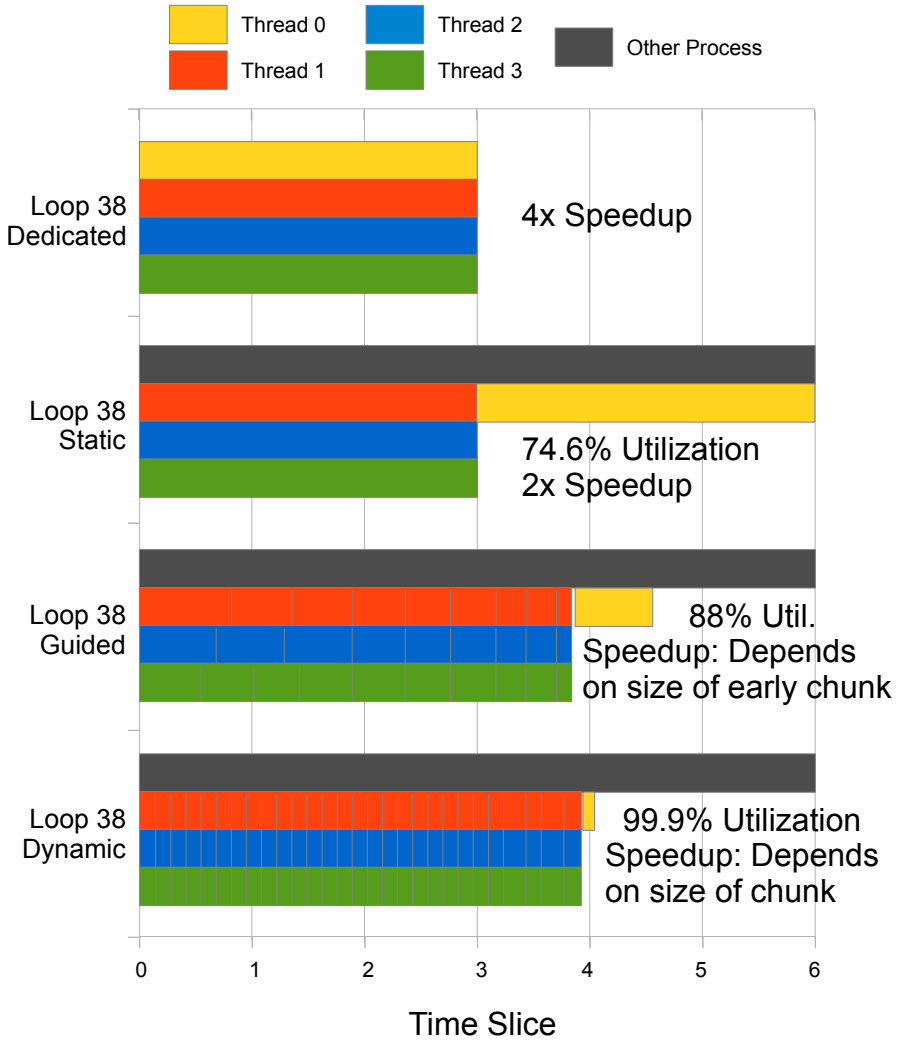


- Process P0: Fixed Priority 2 (infinite loop)
- Process P1: Four threads Fixed Priority 0
- Two P1 threads share one core
- Until one thread is sometimes blocked

OpenMP Dynamic/Guided & Fixed Priority Scheduler

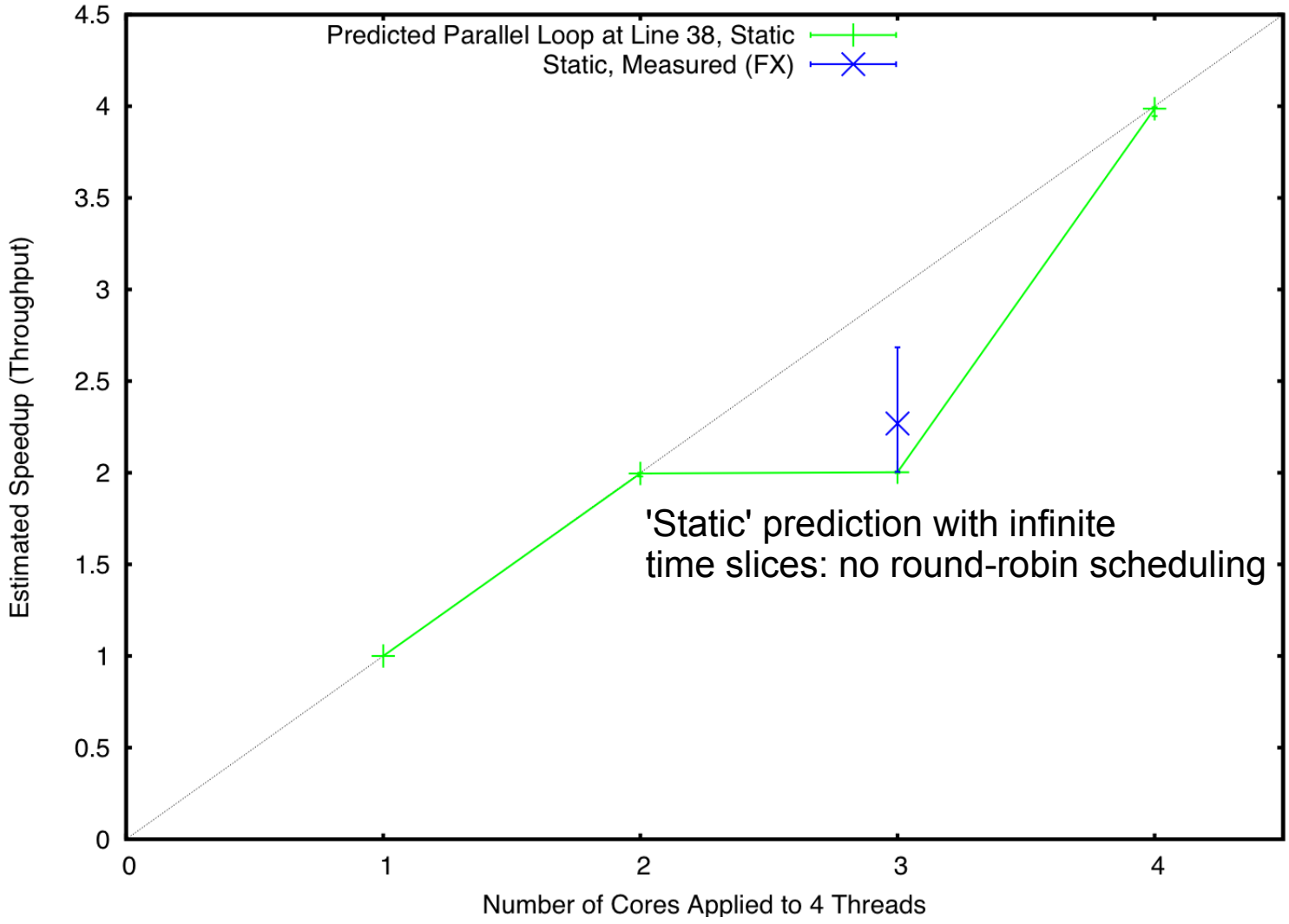
Four Low-Priority Threads & One High-Priority Process Sharing 4 Cores

- High-Priority single-thread consumes one core
- Four threads share other three cores until one is “blocked”
- Static, Guided: potentially large remaining work
- Dynamic: controlled, smaller work chunks



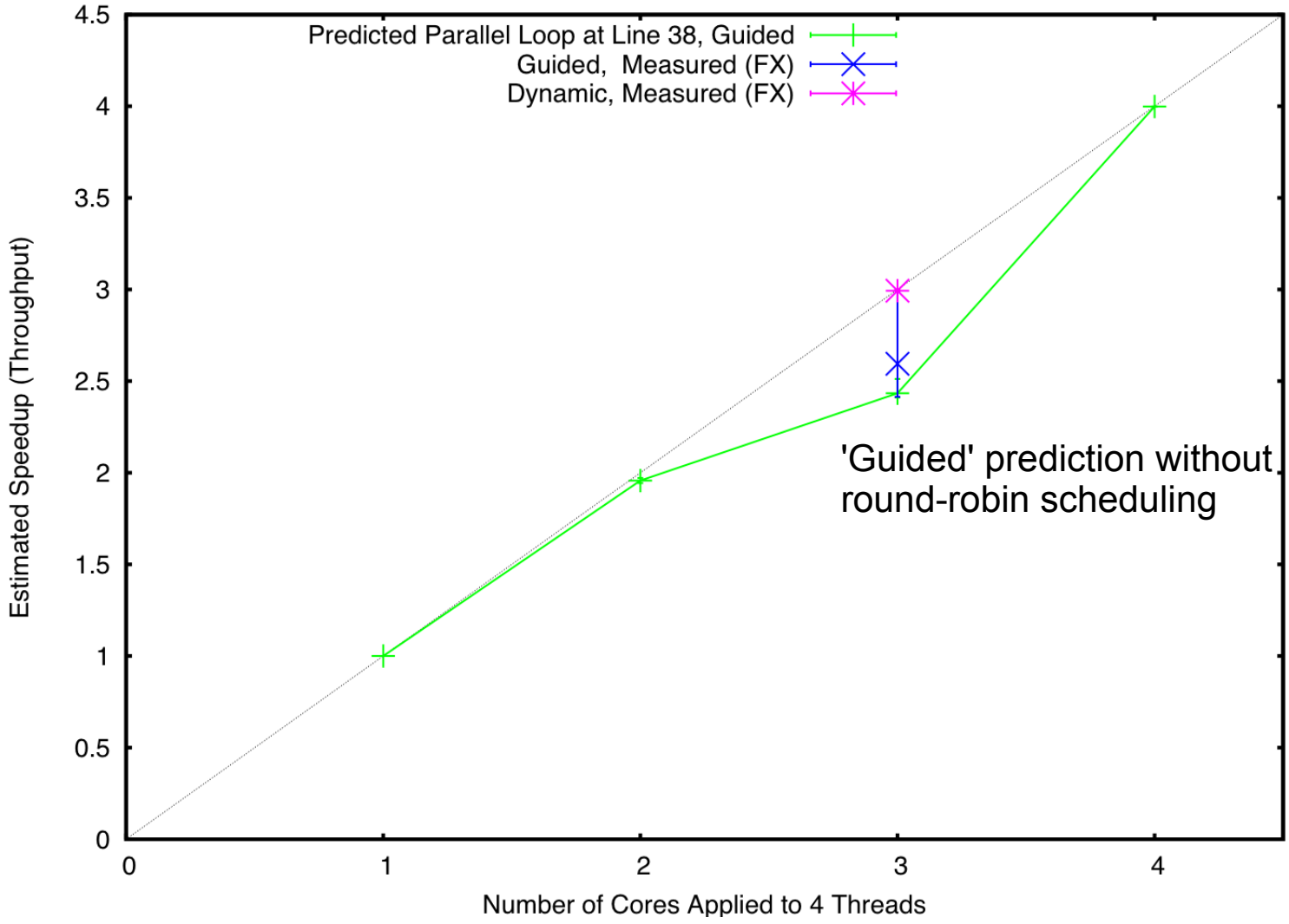
FX Priority Class Experiment: Static

Four Threads run on Three Cores; High-Priority Process on One Core



FX Priority Class Experiment: Guided & Dynamic

Four Threads run on Three Cores; High-Priority Process on One Core



Conclusions: OpenMP Throughput

- OpenMP throughput philosopy:
  Efficiently use *any available* processor cycles
- Tool helps the OpenMP programmer improve scalability
- Tool has uncovered subtle utilization bottlenecks
- Tool can help programmer improve utilization in
  - **Time Share** scheduling class
  - **Fixed Priority** scheduling class with prioritized control
- Dynamic schedule (or tasking): best for utilization
  - If you choose a good chunk size
  - Tool measurements enable good choice

Current Limitations & Future Work

- Nested Parallel Regions
  - Not clear how to interpret speedup
  - Tasking probably superior approach
- Collecting Traces
  - Dropped trace data: high rate of calls, app probably won't scale
  - Very large traces: option to restrict repeated regions; compress
- Threaded & NUMA systems
  - Hardware characteristics: leverage existing tools
  - Tool enables differentiation: OpenMP vs. hardware bottlenecks
  - Add dedicated scaling analysis
- Future Work
  - Larger configurations (in progress)
  - Testing on a broad range of applications


ORACLE®

ORACLE

Mark Woodyard
Principal Software Engineer
mark.woodyard@oracle.com