Memory Placement Optimization (MPO)

Jonathan Chew
Kernel engineer
Solaris Kernel Performance

Overview

- Motivation
- Goals
- Abstraction
- Architecture
- Optimizations
- APIs
- Tools
- Performance
- Status
- Conclusion

Motivation

- Solaris will run on ***Non Uniform Memory Access*** (***NUMA***) machines, but may not perform very well without knowing which CPUs and memory are near each other.
- **Memory Placement Optimization** (**MPO**) project intended to make Solaris aware of NUMA to provide better performance on NUMA machines by optimizing for *locality*

MPO Design Goals

– Enhance performance and reproducibility
– Common framework + means to exploit platform specific features
– Observability
– Benchmarks and techniques for evaluating NUMA performance
– Minimal set of APIs needed for tuning

Abstraction

- *locality group* (*lgroup*)
  – "The *locality group* has been introduced in Solaris to represent the set of CPU-like and memory-like hardware resources which are within some latency of each other."

Lgroups

- Show which CPUs and memory near each other
- Currently use latency as measure of locality, but could include others
- Definition allows for I/O devices
- Hierarchical
  - Necessary to represent more than two levels of locality
  - Can be organized to make it easier to find nearest resources
  - Parents contain resources of children + next nearest resources

How Solaris Uses Lgroups

- Optimize performance via locality
  - Each thread assigned home lgroup on creation
  - Always try to get resources from home and then parent lgroup(s) if necessary
- Optimize for load/bandwidth to ensure lowest latency
  - Spread out threads among lgroups
  - Use random memory allocation for shared memory

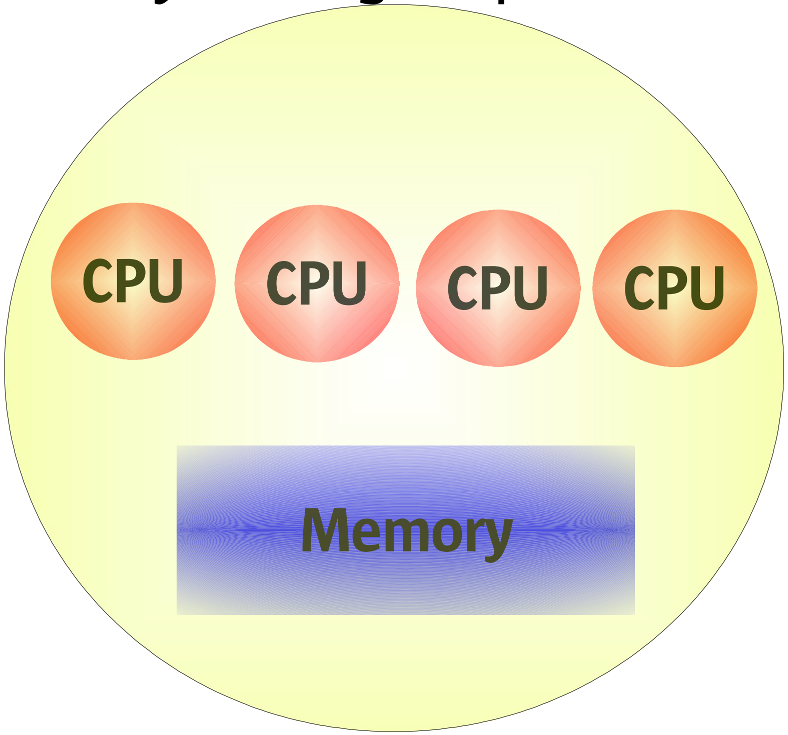
Lgroup Examples

- Abstraction best illustrated by examples given for machines with different memory architectures:
  - Uniform Memory Access (UMA) machine
  - Non-Uniform Memory Access (NUMA) machine (eg Sun Fire V210, V240, V440, 4800, 6800, 6900, 12K, 15K, 20K, 25K)
  - NUMA machine with ring topology (eg. V20z, V40z)

Uniform Memory Access

1 Level of Locality

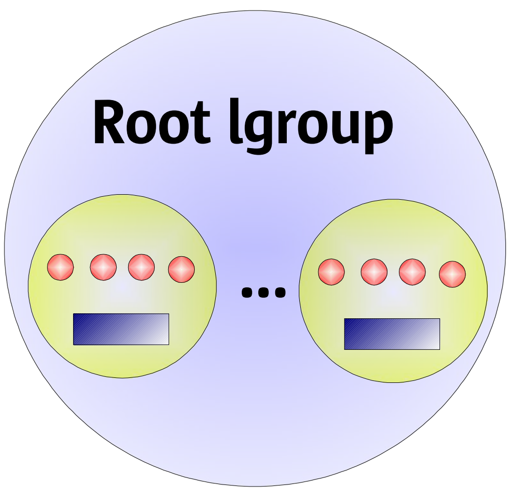
- Same latency between all CPUs and all memory represented by one lgroup

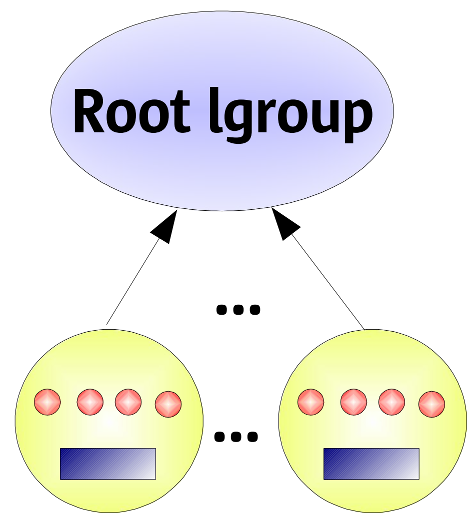


NUMA

2 Levels of Locality

- Local and remote memory latency
- Children lgroups capture CPUs and memory within same local latency of each other
- Root lgroup contains CPUs and memory within remote latency (eg all CPUs and memory)

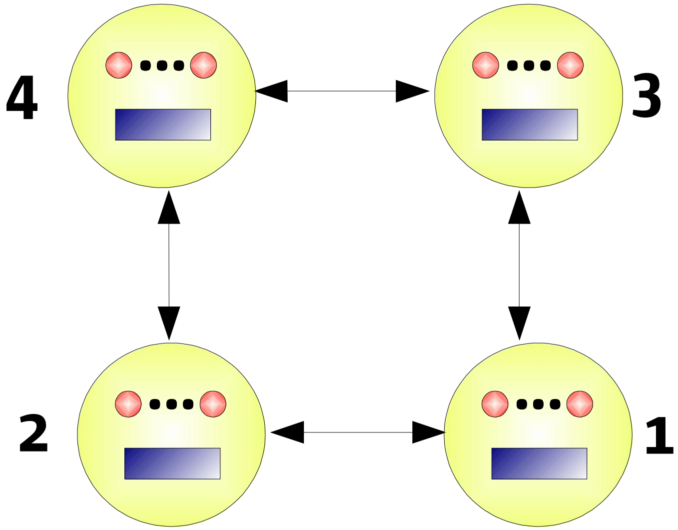




NUMA (ring)

3 Levels of Locality

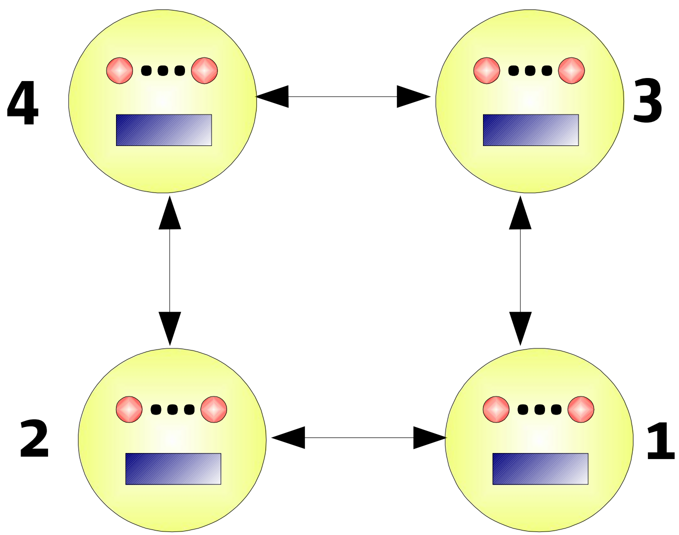
- 4 node ring topology
- Same local latency within each node
- Remote latency determined by sum of cost for each hop needed to reach memory

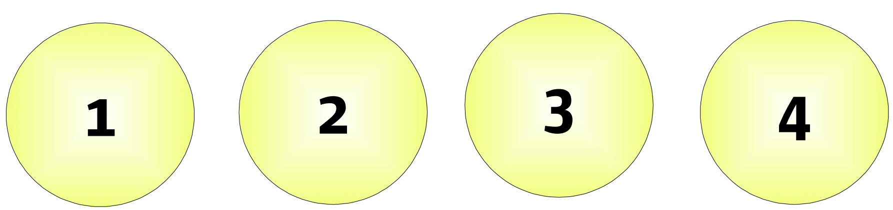


NUMA (ring)

1st Level of Locality

- Lgroups 1, 2, 3, and 4 containing CPU(s) and memory within some local latency

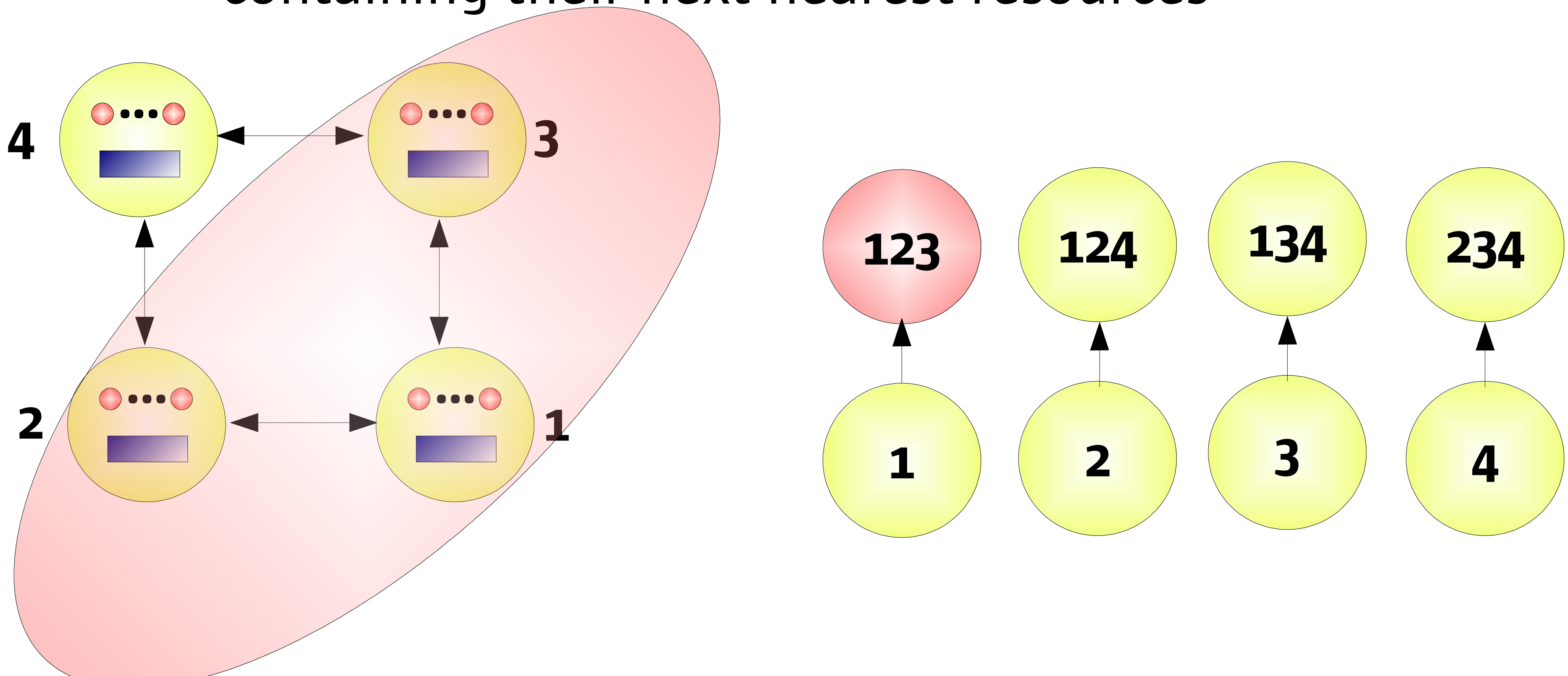




NUMA (ring)

2nd Level of Locality

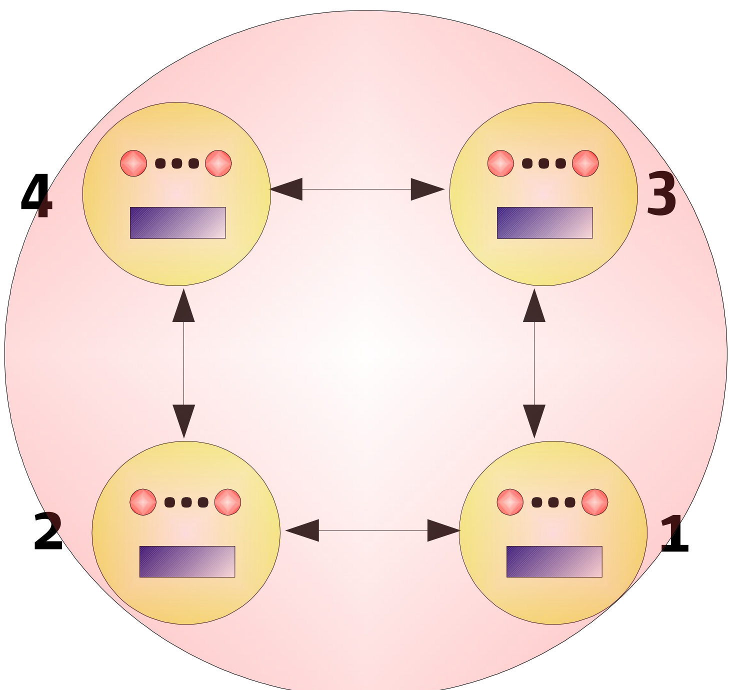
– Lgroups 1, 2, 3, 4, and their parent lgroups containing their next nearest resources

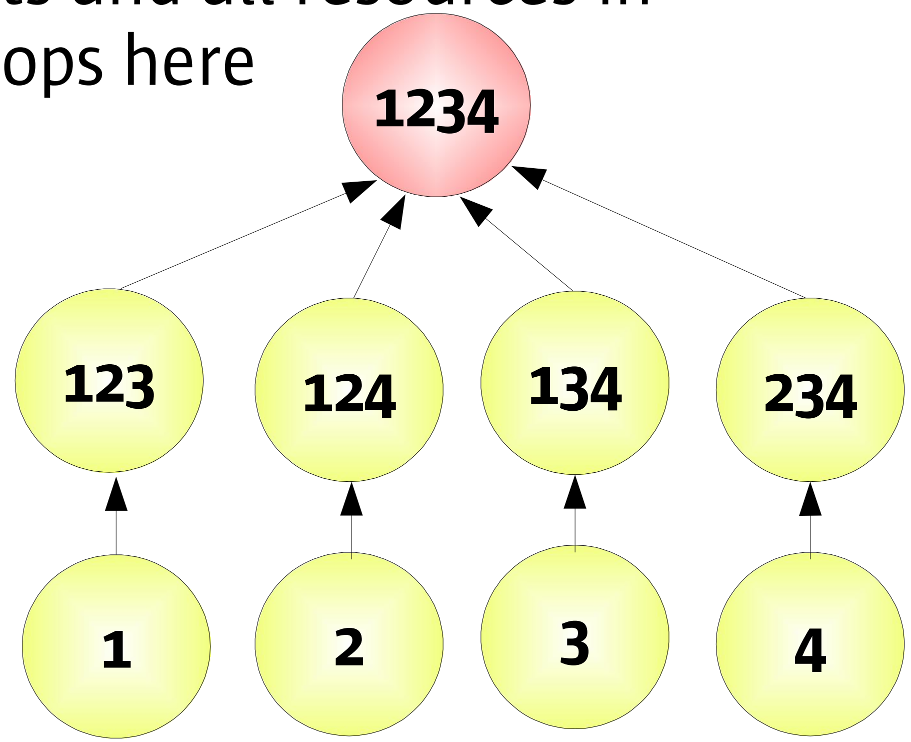


NUMA (ring)

3rd Level of Locality

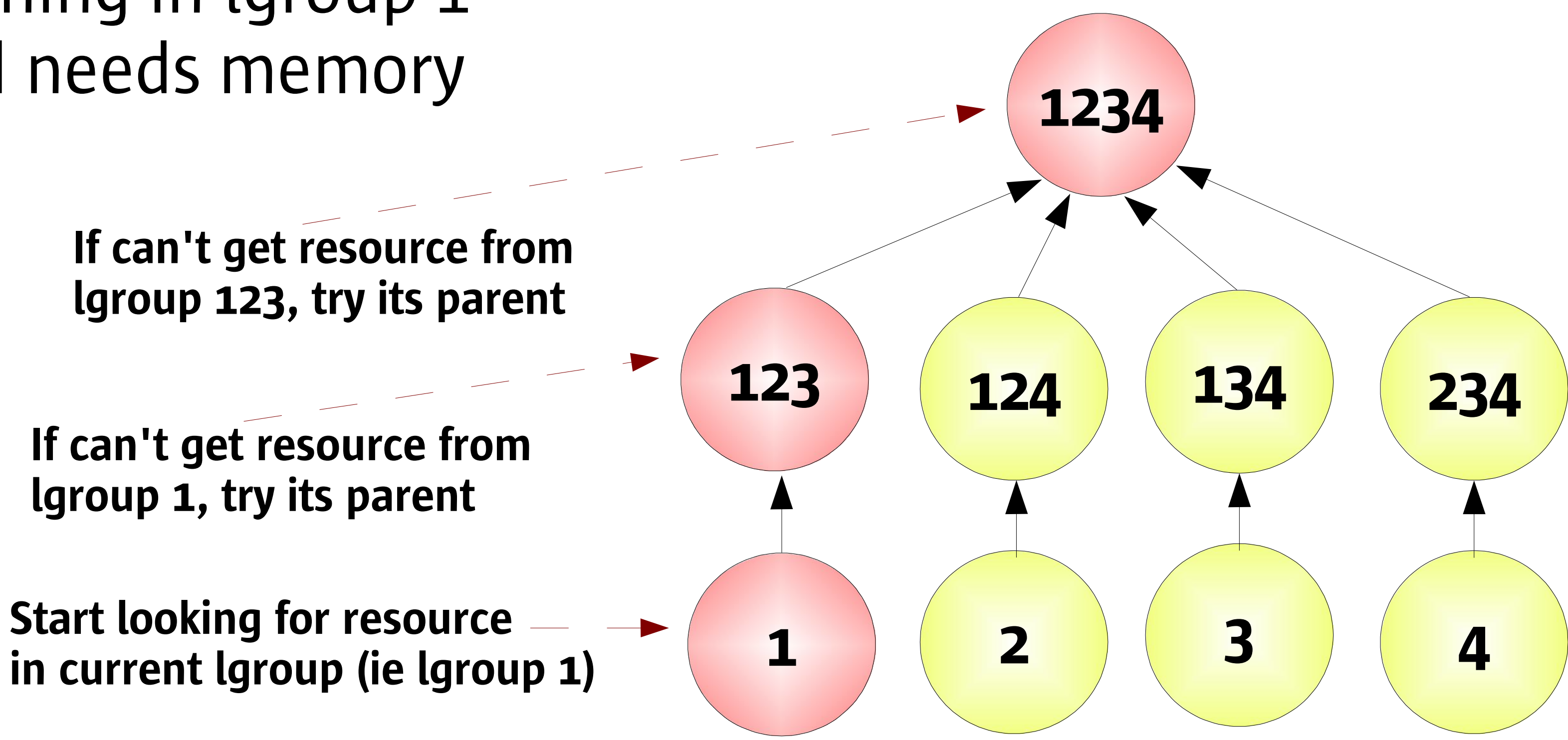
- Root lgroup contains next nearest resources for leaf lgroups' parents and all resources in machine so buck stops here





Finding Resources

- Assume thread is running in lgroup 1 and needs memory



Architecture

– Mostly common kernel code with small, well-defined platform support

Lgroup interface

Common	Platform specific
lgroup management Kstats Memory Thread placement lgroup and lpl topology	Initialization Static memory allocation Hardware configuration

Lgroup creation

– Example of interface between common and platform-specific lgroup support:

void*lgrp_plat_cpu_to_hand(processorid_t);

- Common code asks platform code for platform handle of lgroup containing given CPU ID when CPU comes online
- If platform handle hasn't been seen by common code before, create a new lgroup containing this.

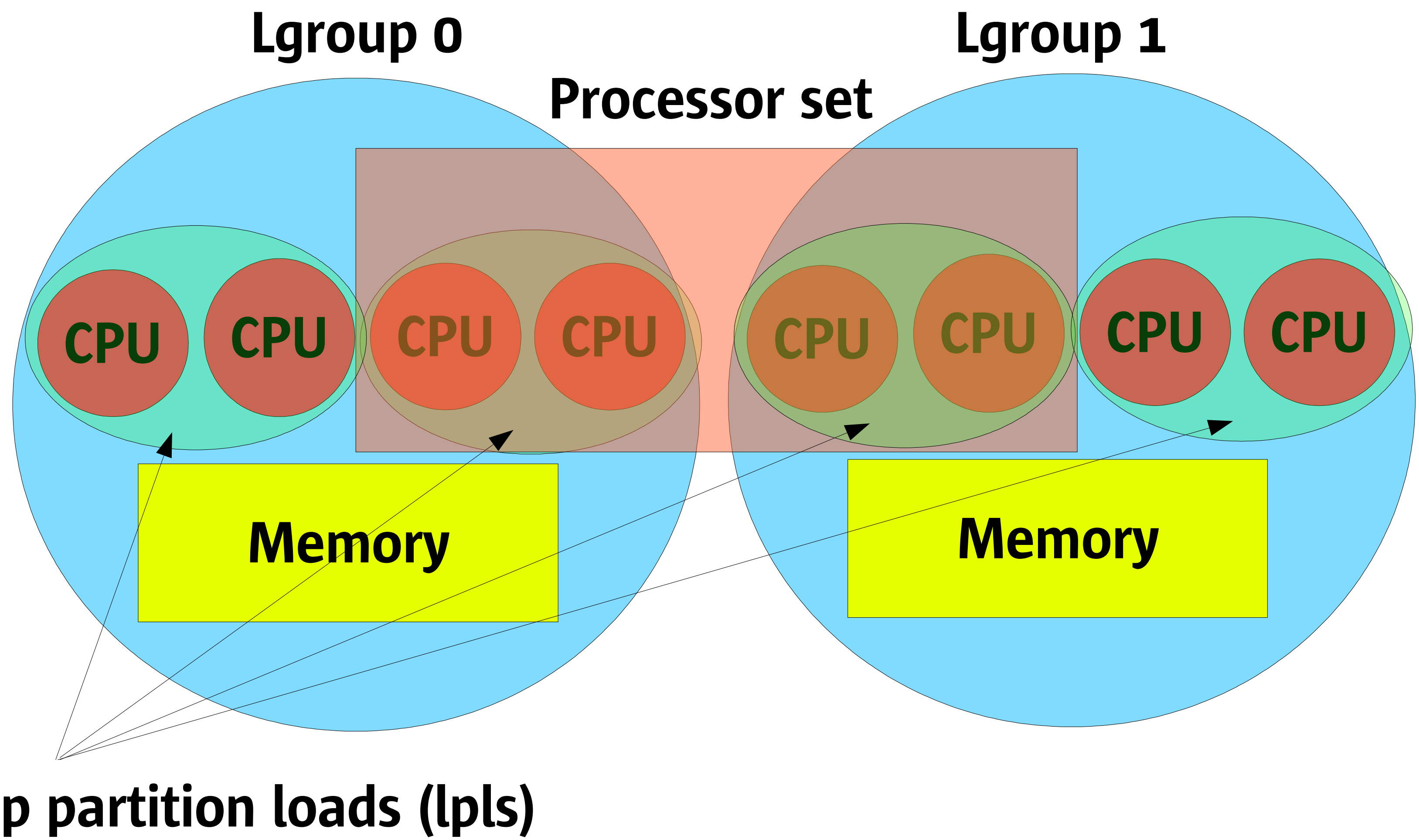
Optimizations

Initial Placement

- Each thread assigned home lgroup upon creation
- Per lgroup load average enables balanced placement
- Anticipated load added to lgroup to avoid piling up on one lgroup
- Platforms can tune how many threads of a process to place on lgroup before trying another one

Lgroup Partition

Lgroup + pset



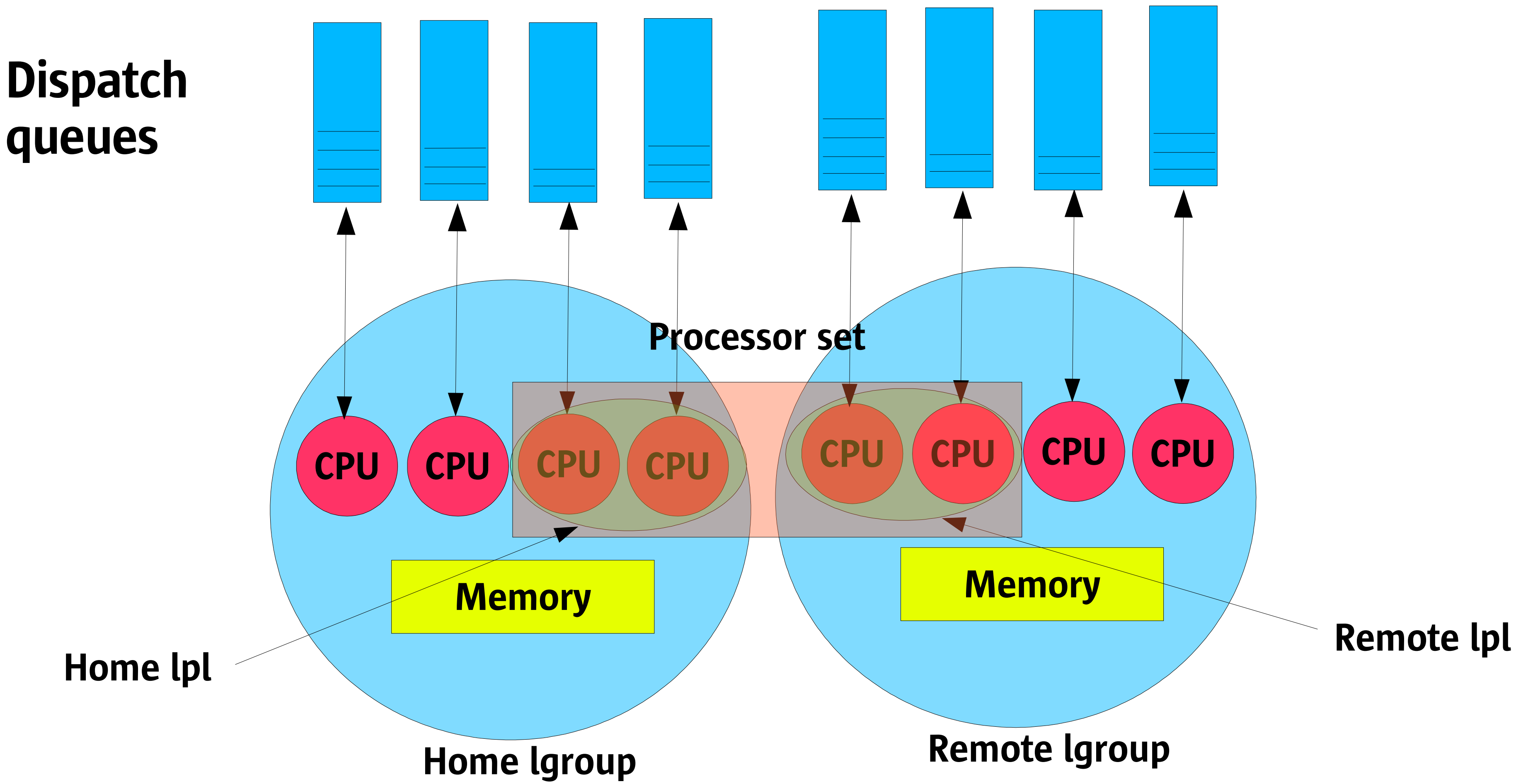
Optimizations

Scheduling

- Always try to schedule thread on its on home unless it can't run there, but will run on a remote lgroup.... Better to run remote than not run at all.
- Balance threads across run queues within lgroup instead of across all run queues
- Idle CPUs prefer to steal from CPUs within same lgroup first
- Use lgroup hierarchy to dispatch or steal to/from nearest CPUs

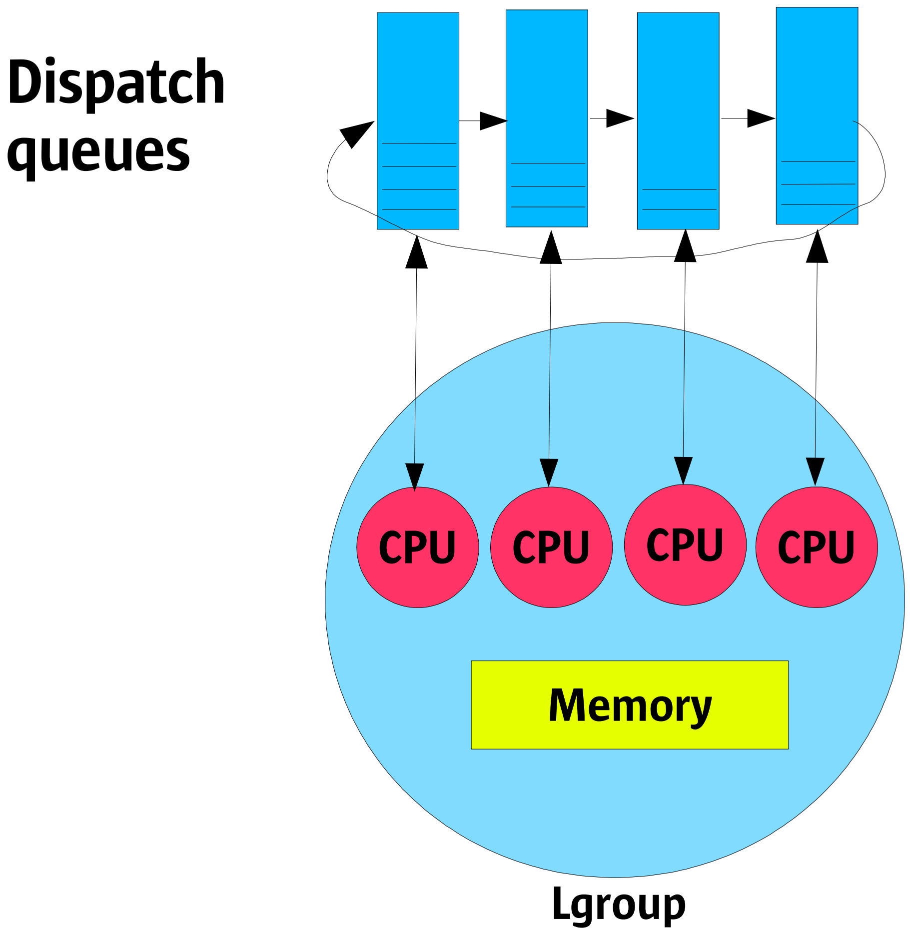
Dispatching

Choose closest CPU with less priority



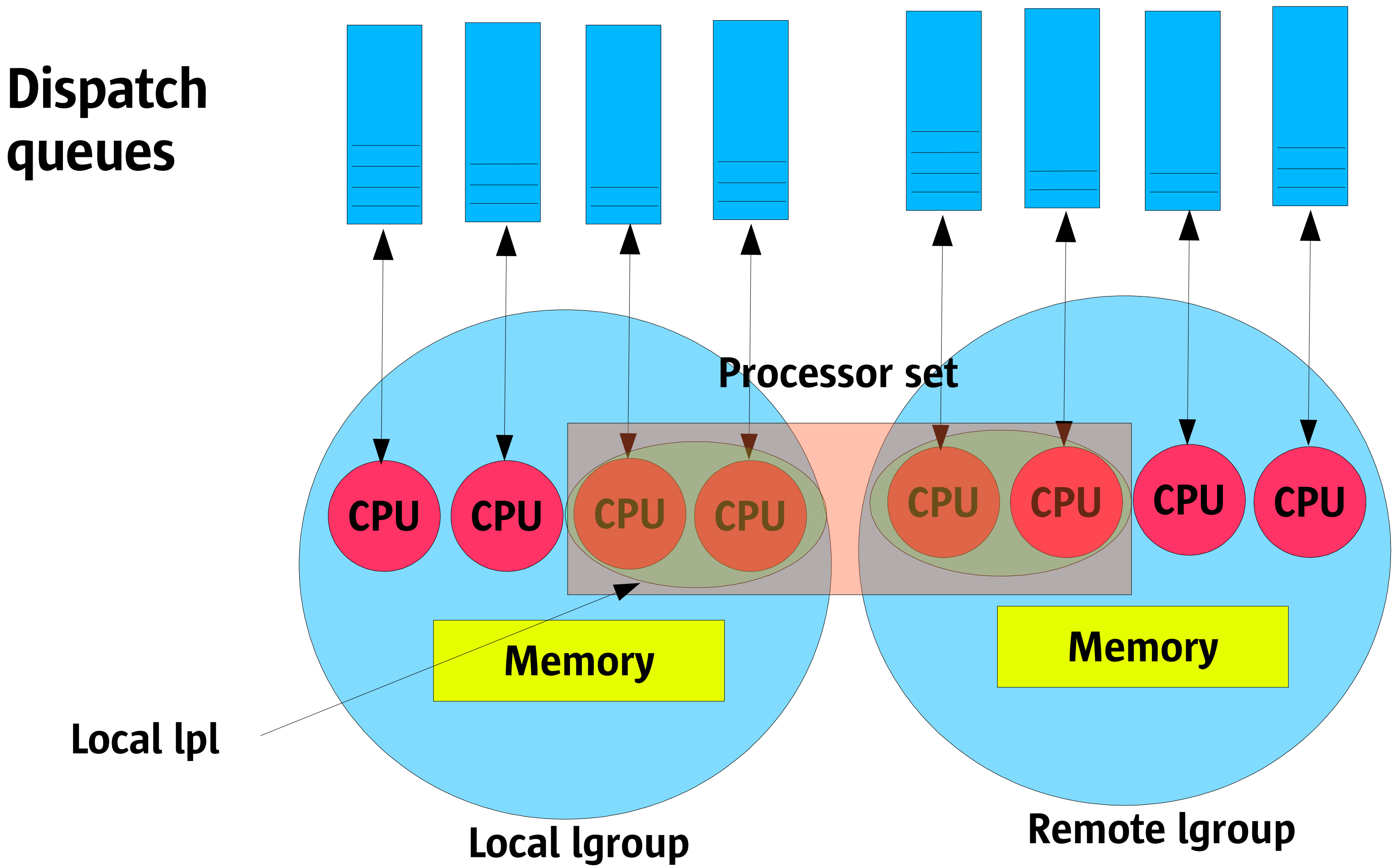
Balancing

Only across CPUs in same lgroup



Stealing

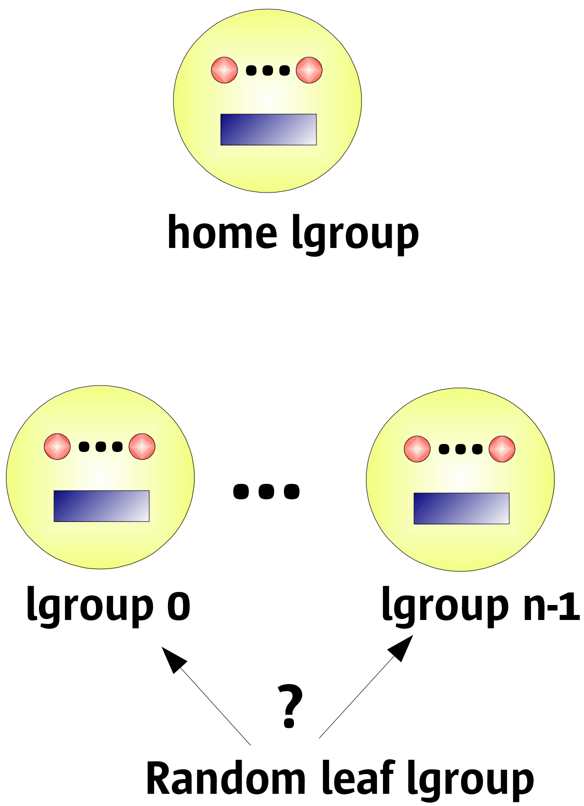
Prefer local lpl over other CPUs in pset



Optimizations

Memory Allocation

- **Next touch**
  - *Allocate memory from home lgroup of faulting thread*
  - *Optimizes for locality*
  - *Default policy for private memory*
- **Random**
  - *Allocate memory randomly from leaf lgroups across domain*
  - *Optimizes for bandwidth*
  - *Default policy for shared memory*



APIs

lgroup

- liblgrp(3LIB)
  - Export lgroup abstraction to user
  - Allows application to do following:
    - Traverse lgroup hierarchy
    - Discover contents of lgroup hierarchy
    - Get/set thread or process' affinity for lgroup

APIs

Memory Placement

- meminfo(2)
  - Allows an application to discover the lgroup containing the physical memory backing given virtual address
- madvise(MADV_ACCESS_*)
  - Default policies work well most of time, but not all.
  - Need way to affect memory allocation
  - *Descriptive* rather than *prescriptive* API
  - May migrate pages to affect placement

madvise(3C)

- **MADV_ACCESS_LWP**
  - *Next LWP to touch specified address range will access it most heavily*
- **MADV_ACCESS_MANY**
  - *Many processes and/or LWPs will access specified address range randomly across machine*
- **MADV_ACCESS_DEFAULT**
  - *Resets kernel's expectation for how specified range will be accessed to default*
- Useful when:
  - Default policy isn't right
  - Want to migrate pages elsewhere as needs change
- Page migration isn't free
  - Cheapest when applied before memory touched
  - Cheaper and better yet to change application to allocate memory efficiently without using madvise(3C)
- LD_PRELOAD library (see madv.so.1(1))

Tools

Existing

- pbind(1M)
  - Can bind and unbind to affect home lgroup
  - Binding takes placement and scheduling out of picture
- madv.so.1(1)
  - LD_PRELOAD library
  - Interposes on memory allocation system calls and calls madvise(MADV_ACCESS_*) on memory
- kstat(1M)
  - *kstat -m lgrp* gives lgroup statistics

Performance

Latency	Bandwidth	Throughput
LmBench latency improves consistently. Reduced latency by ~16% on Sun Fire 6800.	STREAM improves by 300% on 72 CPU Sun Fire E15K	OLTP benchmarks improved by 3-10% on Sun Fire E15K
Swim improves by 40% on Sun Fire E15K		Additional 3-5% improvement for TPC-SO with Oracle 10 using lgroup APIs and madvise(3C)
SPECOMP benchmarks improve by ~0-77% on V40z		Decision support benchmarks got 11% reduction in runtime and 7% more throughput on Sun Fire E15K
		SPECjbb improves by ~7% on V40z

Performance

Hieararchical vs 2 level Lgroup Support on V40z

- SPECjbb
  - ~0-4.6%
- SPECOMP
  - ~0-8% but sometimes worse because of variability
- Linpack
  - ~0-3%

Status

Release	Features
S9	Lgroup common framework, meminfo(2)
S9U1, S10	MPO support for Sun Fire 6800
S9U2, S10	MPO for Sun Fire E15K, madvise(MADV_ACCESS*), madv.so.1
S9U5, S10	Lgroup APIs
S10	MPO support for Sun Blade 2500, V210, V240, V440 MPO 2 level support for Opteron
S10U1, S11	MPO HLS (on Opteron)

Conclusion

- MPO clearly shows benefit on SPARC and Opteron NUMA machines
- Hierarchical Lgroup Support (HLS) improved lgroup framework significantly, but only see small performance gain over 2 level support so far
- Validating and improving lgroup framework with new platforms
- Still more work to do in order to fully achieve goals

Memory Placement Optimization (MPO)

Jonathan.Chew@sun.com

Sun microsystems