Design Principles for End-to-End Multicore Schedulers

Simon Peter* Adrian Schüpbach* Paul Barham†
Andrew Baumann* Rebecca Isaacs† Tim Harris†
Timothy Roscoe*

*Systems Group, ETH Zurich †Microsoft Research

HotPar’10
Context: Barrelyfish Multikernel operating system

- Developed at ETHZ and Microsoft Research
- Scalable research OS on heterogeneous multicore hardware
  - Operating system principles and structure
  - Programming models and language runtime systems
- Other scalable OS approaches are similar
  - Tessellation, Corey, ROS, fos, …
  - Ideas in this talk more widely applicable
Today’s talk topic

OS Scheduler architecture for today’s (and tomorrow’s) multicore machines

▶ General-purpose setting:
  ▶ Dynamic workload mix
  ▶ Multiple parallel apps
  ▶ Interactive parallel apps
Why this is a problem
A simple example

- Run 2 OpenMP applications concurrently
- On 16-core AMD Shanghai system
- Intel OpenMP library
- Linux OS
Why this is a problem
Example: 2x OpenMP on 16-core Linux

- One app is CPU-Bound:
  ```c
  #pragma omp parallel
  for(;;) iterations[omp_get_thread_num()]++;
  ```

- Other is synchronization intensive (eg. BARRIER):
  ```c
  #pragma omp parallel
  for(;;) {
    #pragma omp barrier
    iterations[omp_get_thread_num()]++;
  }
  ```
Why this is a problem

Example: 2x OpenMP on 16-core Linux

- Run for $x$ in $[2..16]$:
  - `OMP_NUM_THREADS=x ./BARRIER &`
  - `OMP_NUM_THREADS=8 ./cpu_bound &`
  - `sleep 20`
  - `killall BARRIER cpu_bound`
- Plot average iterations/thread/s over 20s
Why this is a problem

Example: 2x OpenMP on 16-core Linux

![Graph showing relative rate of progress vs. number of BARRIER threads. The graph compares CPU-bound and BARRIER threads, with the CPU-bound thread maintaining a relatively stable rate of progress as the number of BARRIER threads increases, while the BARRIER thread shows a significant decrease in rate of progress.]
Why this is a problem
Example: 2x OpenMP on 16-core Linux
Why this is a problem
Example: 2x OpenMP on 16-core Linux

Until 8 BARRIER threads
Why this is a problem
Example: 2x OpenMP on 16-core Linux

Until 8 BARRIER threads
CPU-Bound stays at 1
(same thread allocation)
Why this is a problem
Example: 2x OpenMP on 16-core Linux

Until 8 BARRIER threads
CPU-Bound stays at 1 (same thread allocation)
BARRIER degrades (due to increasing cost)
Why this is a problem
Example: 2x OpenMP on 16-core Linux

Until 8 BARRIER threads
CPU-Bound stays at 1 (same thread allocation)
BARRIER degrades (due to increasing cost)
Space-partitioning
Why this is a problem
Example: 2x OpenMP on 16-core Linux

From 9 threads (threads > cores)
Time-multiplexing
Why this is a problem
Example: 2x OpenMP on 16-core Linux

From 9 threads (threads > cores)
Time-multiplexing
CPU-Bound degrades linearly
Why this is a problem

Example: 2x OpenMP on 16-core Linux

From 9 threads (threads > cores)

Time-multiplexing

CPU-Bound degrades linearly

BARRIER drops sharply (only makes progress when all threads run concurrently)
Why this is a problem

Example: 2x OpenMP on 16-core Linux

- Gang scheduling or smart core allocation would help

- Gang scheduling:
  - OS unaware of apps’ requirements
  - The run-time system could’ve known
    - Eg. via annotations or compiler

- Smart core allocation:
  - OS knows general system state
  - Run-time system chooses number of threads

- Information and mechanisms in the wrong place
Why this is a problem
Example: 2x OpenMP on 16-core Linux

Huge error bars (min/max over 20 runs)
Random placement of threads to cores
Why this is a problem
16-core AMD Shanghai system

- Same-die L3 access twice as fast as cross-die
- OpenMP run-time does not know about this machine
Why this is a problem
16-core AMD Shanghai system

- Same-die L3 access twice as fast as cross-die
- OpenMP run-time does not know about this machine
Why this is a problem
16-core AMD Shanghai system

- Same-die L3 access twice as fast as cross-die
- OpenMP run-time does not know about this machine
Why this is a problem

Example: 2x OpenMP on 16-core Linux

2 threads case: Performance difference of 0.4
Why this is a problem
System diversity

### AMD Opteron (Magny-Cours)
- On-chip interconnect

### Sun Niagara T2
- Flat, fast cache hierarchy

### Intel Nehalem (Beckton)
- On-die ring network
Why this is a problem

System diversity

- AMD Opteron (Magny-Cours)
  - On-chip interconnect
  - Manual tuning increasingly difficult
  - Architectures change too quickly
  - Offline auto-tuning (eg. ATLAS) limited

- Sun Niagara T2
  - Flat, fast cache hierarchy

- Intel Nehalem (Beckton)
  - On-die ring network
Online adaptation remains viable
- Easier with contemporary runtime systems
  - OpenMP, Grand Central Dispatch, ConcRT, MPI, …
  - Synchronization patterns are more explicit
- But needs information at right places
The end-to-end approach

► The system stack:

<table>
<thead>
<tr>
<th>Component</th>
<th>Related work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Heterogeneous, …</td>
</tr>
<tr>
<td>OS scheduler</td>
<td>CAMP, HASS, …</td>
</tr>
<tr>
<td>Runtime systems</td>
<td>OpenMP, MPI, ConcRT, McRT, …</td>
</tr>
<tr>
<td>Compilers</td>
<td>Auto-parallel., …</td>
</tr>
<tr>
<td>Programming paradigms</td>
<td>MapReduce, ICC, …</td>
</tr>
<tr>
<td>Applications</td>
<td>annotations, …</td>
</tr>
</tbody>
</table>

► Involve all components, top to bottom
► Need to cut through classical OS abstractions
► Here we focus on OS / runtime system integration
Design Principles
Design principles

1. Time-multiplexing cores is still needed

- Resource abundance ≠ scheduler freedom
- Asymmetric multi-core architectures
  - Contention for “big” cores
- Provide real-time QoS to interactive apps, not wasting cores
  - Avoid power wasted through over-provisioning
Design principles

2. Schedule at multiple timescales

- Interactive workloads are now parallel
  - Requirements might change abruptly
  - Eg. parallel web browser
- Much shorter, interactive time scales
- Thus need **small overhead** when scheduling
  - Synchronized scheduling on every time-slice won’t scale
Implementation in Barrelfish

- Combination of techniques at different time granularities
  - Long-term placement of apps on cores
  - Medium-term resource allocation
  - Short-term per-core scheduling
Implementation in Barrelfish

- Combination of techniques at different time granularities
  - Long-term placement of apps on cores
  - Medium-term resource allocation
  - Short-term per-core scheduling
- Phase-locked gang scheduling
  - Gang scheduling over interactive timescales
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

Best-effort (actual trace):

Phase-locked gang scheduling (actual trace):
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

![Diagram showing phase-locked gang scheduling]

Best-effort (actual trace):

Progress only in small time windows

Phase-locked gang scheduling (actual trace):
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

**Best-effort (actual trace):**

**Phase-locked gang scheduling (actual trace):**

Synchronize core-local clocks
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

Best-effort (actual trace):

Phase-locked gang scheduling (actual trace):

Agree on future gang start time
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

Best-effort (actual trace):

Phase-locked gang scheduling (actual trace):

...and gang period
Phase-locked gang scheduling

- Decouple schedule synchronization from dispatch

Best-effort (actual trace):

Phase-locked gang scheduling (actual trace):

Resync in future when necessary
Design principles

3. Reason online about the hardware

- We employ a **system knowledge base**
  - Contains rich representation of the hardware
  - Queries in subset of first-order logic
  - Logical unification aids dealing with diversity

- Both OS and apps use it
Design principles
4. Reason online about each application

- OS should exploit knowledge about apps for efficiency
  - Eg. gang schedule threads in an OpenMP team
  - But no sense in gang scheduling unrelated threads
- A single app might go through different phases
  - Optimal allocation of resources changes over time

Implementation:
- Apps submit scheduling manifests to planner
  - Contain predicted long-term resource requirements
  - Expressed as constrained cost-functions
  - May make use of any information in the SKB
Design principles
5. Applications and OS must communicate

- Implementing the end-to-end principle
- Resource allocation may be renegotiated during runtime

Implementation:
- Hardware threads run user-level dispatchers
  - Cf. Psyche, inheritance scheduling
- Related dispatchers are grouped into dispatcher groups
  - Derived from RTIDs of McRT
  - Used as handles when renegotiating
- Scheduler activations [Anderson 1992] to inform app
Implementation in the Barrelfish OS
Open questions

- What are appropriate mechanisms and timescales for inter-core phase synchronization?
- How can programmers provide useful concurrency information to the runtime?
- How efficiently can runtime specify requirements to OS?
- Hidden cost (if any) of decoupling scheduling timescales?
- Tradeoffs between centralized and distributed planners?
- Appropriate level of expressivity for the SKB?