Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn
Systems Technology Lab
Intel Corporation

Jae Moon Seo
jmseo@rtcc.hanyang.ac.kr
Real-Time Computing and Communications Lab
Hanyang University
Contents

- Introduction
- AMPS
  - Design Space
  - Asymmetry-Aware Load Balancing
  - Faster-Core-First Scheduling
  - NUMA-Aware Thread Migration
- Evaluation
- Conclusion
Performance-asymmetric architectures

- A processor contains multiple cores with the same instruction set but different performance characteristics.
- Higher performance at lower costs in terms of die area and power consumption.
Introduction

What is the problem?
- Operating systems traditionally assume homogeneous hardware
- OS schedulers do not directly work well on asymmetric architectures

Problem solving
- OS must take into account hardware asymmetry when making scheduling decisions

AMPS
AMPS
- An asymmetric multiprocessor scheduler
- Support both SMP-and NUMA-style architectures

Three components in AMPS
- Asymmetry-aware load balancing
- Faster-core-first scheduling
- NUMA-aware migration
Design Space

- **Distributed run-queue model**
  - Per core has its own run queue
  - Periodically, it balances the load on every core

- **Thread-independent policies**
  - Schedule threads independently regardless of application types and dependencies.
Goals of AMPS

☐ Performance
  ▪ Most applications should achieve good performance

☐ Fairness
  ▪ Threads with same priority should receive same core processing.

☐ Repeatability
  ▪ Different runs of an application under similar conditions should have similar performance.
Asymmetry-Aware Load Balancing

- Quantifying Core Computing Power
  - Core’s scaled computing power (P)
    - It computing power divided by the system’s minimum core computing power.
    - In this paper we use core frequency.

- Scaled Load
  - We define its scaled load, L, to be its run queue length divided by P.

- Load Balancing
  - In the system if \( L_{\text{max}} - L_{\text{min}} \leq 1 \), we say it is load-balanced.
Example

- \( L_0 = \frac{4}{2} = 2, \ L_1 = \frac{2}{1} = 2, \ L_2 = \frac{2}{1} = 2, \ L_3 = \frac{2}{1} = 2 \)

- \( L_{\text{max}} - L_{\text{min}} = 2 - 2 = 0 \leq 1 \)

- So it is load balancing.
Faster-Core-First Scheduling

- Goal of Faster-Core-First Scheduling
  - Threads move to and run on a faster core as long as the core is under-utilized.

- Initial thread placement:
  - For a new thread, we compute the new scaled load for each core assuming thread would run on it.
  - Then we choose core with the minimum new scaled load.
  - If tied, we choose the faster core.

- Dynamic thread migration.
  - AMPS allowing threads to migrate to cores that have lower scaled load even if their original cores can become idle.
Motivating Example

Two options

1) Moving one thread to the fast core so that it runs 4 times faster
2) Moving all three threads to it so that each could run 4/3 times faster.
Discussion about Faster-Core-First Scheduling

- Thread migrates incur compulsory cache misses
  - Overhead associated with a thread’s migration is **negligible** in SMP systems
  - But can be **significant** in NUMA systems

We need NUMA-aware migration policies
A four-core NUMA system

- Cores connected to the same memory controller are in the same node.
- Within the same node, every core is equidistant to local memory.
NUMA-Aware Thread Migration

- Overhead of migrates
  - Software overhead
    - The time to move the thread from A to B.
    - The overhead of these steps is mostly negligible.
  - Hardware overhead
    - The thread incurs extra compulsory misses in cache-type structures,
    - Penalties of missing in these structures can be higher after the migration due to NUMA.
NUMA-Aware Thread Migration

How can we known overhead is high?

- The thread incurs a high number of LLC (Last-level cache) misses after the migration.
- A large fraction of these LLC misses require remote memory access.
NUMA-Aware Thread Migration

Predicting Migration Overhead

- Key idea
  - Keep track of the working set of each thread on each node.

Two approaches

- Working Sets
  - Unix systems maintain system-wide page Least Recently Used (LRU) lists, which approximate a global working set for all processes.

- Resident Sets
  - The resident set of a thread (or process) includes its pages that are currently in memory.
  - Most OSes maintain the resident set size (RSS) for each process.
  - AMPS can track per-thread, per-node resident sets.
NUMA-Aware Thread Migration

Prediction algorithm

- If all of these conditions are true, we say its overhead is high.
  1. Core A and core B are in different nodes.
  2. Core A is in a node for which thread T has the maximum RSS counter value compared to all other nodes.
  3. The RSS counter value of thread T for core A’s node is greater than the LLC size of core B.
Running Thread Migration Policies

- The Always policy
  - AMPS always allows a running thread to migrate

- The Same-Node policy
  - AMPS allows a running thread to migrate only within the same node.

- The RSS policy.
  - It disallows a running thread to migrate if it predicts the migration overhead to be high.
EVALUATION

- System configuration

<table>
<thead>
<tr>
<th>Features</th>
<th>SMP</th>
<th>NUMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Processors/node</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Cores/processor</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total cores</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>L1 D-cache/core</td>
<td>16 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>L2 cache/core</td>
<td>1 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>L3 cache/core</td>
<td>none</td>
<td>8 MB</td>
</tr>
<tr>
<td>L4 cache/node</td>
<td>none</td>
<td>256 MB (DDR2)</td>
</tr>
<tr>
<td>Memory/node</td>
<td>8 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Total memory</td>
<td>8 GB</td>
<td>64 GB</td>
</tr>
<tr>
<td>Memory latency</td>
<td>79 ns</td>
<td>local: 68 ns 1-hop: 146 ns 2-hop: 176 ns</td>
</tr>
</tbody>
</table>

- Methodology

- For both SMP and NUMA systems we are making 75% cores’ frequencies 50% lower than others.
EVALUATION

SMP Evaluation

Better Performance
EVALUATION

- **SMP Evaluation**

More Fairness
EVALUATION

- SMP Evaluation

(a) Stock Linux results (official score 29720).

(b) AMPS results (official score 29840).

More Repeatability
EVALUATION

NUMA Evaluation

(a) NUMA-1 performance results.

Table 4: NUMA-1 number of migrations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Stock Linux</th>
<th>AMPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Always</td>
<td>Same-Node</td>
</tr>
<tr>
<td>Ammp</td>
<td>62</td>
<td>8,947</td>
</tr>
<tr>
<td>Applu</td>
<td>116</td>
<td>95,942</td>
</tr>
<tr>
<td>Apsi</td>
<td>211</td>
<td>47,177</td>
</tr>
<tr>
<td>Art</td>
<td>27</td>
<td>86</td>
</tr>
<tr>
<td>Equake</td>
<td>66</td>
<td>174,265</td>
</tr>
<tr>
<td>Fma3d</td>
<td>272</td>
<td>91,520</td>
</tr>
<tr>
<td>Gafort</td>
<td>179</td>
<td>46,621</td>
</tr>
<tr>
<td>Galgel</td>
<td>108</td>
<td>80,178</td>
</tr>
<tr>
<td>Mgrid</td>
<td>94</td>
<td>457,378</td>
</tr>
<tr>
<td>Swim</td>
<td>217</td>
<td>79,278</td>
</tr>
<tr>
<td>Wupwise</td>
<td>238</td>
<td>102,744</td>
</tr>
</tbody>
</table>

Better Performance
EVALUATION

NUMA Evaluation

(b) NUMA-2 performance results.

Better Performance
Conclusion

- **AMPS OS scheduler**
  - It can efficiently manages SMP and NUMA-style performance asymmetric architectures.

- **Benefit of AMPS scheduler.**
  - Performance
  - Fairness
  - Repeatability