Lottery Scheduling:
Flexible Proportional-Share Resource Management

Carl A. Waldspurger  William E. Weihl
MIT Laboratory for Computer Science

OSDI ’94

2016-03-28
Presented by Lee Dong Yun, Jo Hyung Il, Nam Taek Eun
Contents

• Background
• Overview
• Design & Implementation
• Evaluation
• Contribution
• Flaws
Background (1)

• Scheduler needs
  – Flexibility
  – High responsiveness
  – Modular
  – Robust
  – Fine grained
  – Low-overhead
  – Conceptually-simple
Background (2)

• Priority based scheduling is often ad-hoc[1]
  – Pretty good for considering all the factors
• Several problems of priority scheduling
  – Absolute priority schemes
  – Fairness
  – Too many parameters make scheduler complicated
• Proportional-share scheduling in network packet scheduling gets famous

Approach to Solve Fairness\textsuperscript{[1,2]}

• Introduce dynamically controlled priority
• Solve absolute priority problem
• Limitations
  – Large overhead
  – Response time is bad

\textsuperscript{[1]} G. J. Henry. “The Fair Share Scheduler”, AT&T Bell Lab, 1984
Proportional-Share Scheduling

- At that time, main issue in packet scheduling in network is proportional-share scheduling
  - GPS is ideal model
    - Importance of fair queueing
  - Problem of WFQ\textsuperscript{[1]}
    - Monitoring overhead was high

Overview

• Lottery scheduling
  – Randomized resource allocation mechanism
  – Proportional-share resource management

• Guarantee probabilistic fairness
  – Starvation-free
  – Low overhead than others

• Simple and elegant with not that much overhead
Design: Lottery

• In each round, one winner is picked
• Thread with winning ticket gets CPU resource
• Several implementations can exist
  – List based structure
  – Tree based structure
Design: Lottery Tickets

- Each thread receives variable numbers of lottery tickets
- Amount
  - The number of tickets that received
- Currency
  - Who published lottery tickets?
Design: Ticket Currency

- Looks like local lottery at each level
- Makes trusty boundaries
- Global load balancing
Design: Balancing Issue (1)

• **Ticket transfers**
  – One thread can send a bunch of tickets to another
  – Useful in interactive systems
  – Eliminate priority inversion problem

• **Ticket inflation & deflation**
  – If more tickets are added, the value of one ticket decreases
  – With trusting threads, it can be the key to dynamic balancing between threads
**Design: Balancing Issue (2)**

- **Ticket compensation**
  - Threads which consumes a fraction $f$ of time quantum receives $1/f$ times scheduling until they get lottery

- **Use whole time quantum**
- **Tickets : 400**

- **Use 1/5 time quantum**
- **$f = 1/5$**
- **Tickets : 400 -> 2000 ($1/f$)**
Experiment Environment

• Mach 3.0 microkernel (MK82)
• 25MHz MIPS-based DEC station 5000/125
• Apps
  – Dhrystone benchmark
  – Monte-Carlo integration
  – Multithreaded client-server
  – MPEG Video
Evaluation

Fairness Over Time
- 2 : 1 ticket allocation for 2 threads

Controlling Video Rates
- 3 video threads with different ticket
- At arrow point, change ticket ratio
Contributions

• Simple, flexible, but strong scheduler
  – Solve priority inversion and starvation
  – Fairness without much overhead
  – Dynamically flexible scheduling

• Well support for modular management
  – Can adjust resource allocation without explicit communication by currencies, ticket inflation and deflation
Flaws

• Bad response time
  – Do not provide responsiveness for interactive systems

• Non-deterministic
  – Unsuitable when the programmers may control better

• Probabilistic can not guarantee when the universe of ticket is small or too big

• Is it really ‘random’ number generator?
  – If skewed, it’s critical to entire system
After Lottery Scheduling (1)

• Stride scheduling\[1\]
  – Select the smallest value
  – Pass is advanced by its stride value
  – Ties are broken arbitrarily

\[1\] C. A. Waldspurger, “Lottery and stride scheduling”, MIT Lab of Computer Science, 1995
After Lottery Scheduling (2)

- Stride scheduling[1]
  - Task $\tau_1$ : tickets = 3, stride = 2
  - Task $\tau_2$ : tickets = 2, stride = 3
  - Task $\tau_3$ : tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

• Stride scheduling[1]
  – Task $\tau_1$: tickets = 3, stride = 2
  – Task $\tau_2$: tickets = 2, stride = 3
  – Task $\tau_3$: tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

- Stride scheduling[1]
  - Task $\tau_1$: tickets = 3, stride = 2
  - Task $\tau_2$: tickets = 2, stride = 3
  - Task $\tau_3$: tickets = 1, stride = 6

<table>
<thead>
<tr>
<th>$\tau_1$ pass value</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\tau_2$ pass value</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$\tau_3$ pass value</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
</table>

After Lottery Scheduling (2)

• Stride scheduling[1]
  – Task $\tau_1$ : tickets = 3, stride = 2
  – Task $\tau_2$ : tickets = 2, stride = 3
  – Task $\tau_3$ : tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td></td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

- Stride scheduling\[1\]
  - Task $\tau_1$ : tickets = 3, stride = 2
  - Task $\tau_2$ : tickets = 2, stride = 3
  - Task $\tau_3$ : tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

- **Stride scheduling**[1]
  - Task $\tau_1$: tickets = 3, stride = 2
  - Task $\tau_2$: tickets = 2, stride = 3
  - Task $\tau_3$: tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td></td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

- **Stride scheduling**[1]
  - Task $\tau_1$ : tickets = 3, stride = 2
  - Task $\tau_2$ : tickets = 2, stride = 3
  - Task $\tau_3$ : tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Lottery Scheduling (2)

- **Stride scheduling**\(^1\)
  - Task \(\tau_1\): tickets = 3, stride = 2
  - Task \(\tau_2\): tickets = 2, stride = 3
  - Task \(\tau_3\): tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_1) pass value</td>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\tau_2) pass value</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\tau_3) pass value</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) C. A. Waldspurger, “Lottery and stride scheduling”, MIT Lab of Computer Science, 1995

After Lottery Scheduling (2)

• Stride scheduling\[1\]
  – Task $\tau_1$ : tickets = 3, stride = 2
  – Task $\tau_2$ : tickets = 2, stride = 3
  – Task $\tau_3$ : tickets = 1, stride = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$ pass value</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_2$ pass value</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$\tau_3$ pass value</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

Question?
Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors

David Tam, Reza Azimi, Michael Stumm

University of Toronto

SIGOPS ’07
Background

- Potential improvement in clock speed diminished
- Improving clock speed → Increasing parallelism (multi-core)
  - IBM Power 5, 2004
  - Intel Core 2 Duo, 2006
- SMT, CMP, SMT Technologies are introduced for parallelism
SMP-CMP-SMT?

- **SMP** (Symmetric Multiprocessing)
  - $\geq 2$ homogeneous processors
  - Single shared main memory and system bus
  - Single OS that treats all processors equally

- **CMP** (Chip Multiprocessing)
  - Some cores onto same chip

- **SMT** (Simultaneous Multithreading)
  - $\geq$ threads on a single CPU core
  - ex. Intel hyper threading
SMP-CMP-SMT ?

IBM Power 5 system
Problem

- New performance problems arose with changed system.
  - Remote cache access overhead
- Existing schedulers were based on a single CPU core.
- New concept of scheduling is required → thread clustering
Remote cache access overhead

• Sharing L1 cache. 1~2 cycles latency
Remote cache access overhead

- Sharing L2 cache. 10~20 cycles latency
Remote cache access overhead

- Accessing remote cache. 100 ~ cycles latency
Remote cache access overhead

- Accessing remote cache. 100 ~ cycles latency
Remote cache access overhead

- Non-uniform data sharing latency

<table>
<thead>
<tr>
<th>Two threads in</th>
<th>Data shared at</th>
<th>Data share latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same core</td>
<td>L1 cache</td>
<td>1 ~ 2</td>
</tr>
<tr>
<td>Same chip</td>
<td>L2 cache</td>
<td>10 ~ 20</td>
</tr>
<tr>
<td>Different chip</td>
<td>Remote L2 cache</td>
<td>100 ~</td>
</tr>
</tbody>
</table>

- Solution
  - Detecting heavy sharing threads and clustering that threads
  - Locating cluster of threads onto same chip
Implementation of scheduling

1. Monitoring stall breakdown due to remote cache access
2. Detecting sharing patterns
3. Thread clustering
4. Thread Migration
Monitoring stall breakdown

- PMU can observe micro-architectural events in processor
- Observe remote cache access to find cross-chip communication.
- If 20% of cycles are spent for accessing remote cache, start detecting sharing patterns
Detecting sharing patterns

Thread

<table>
<thead>
<tr>
<th>Vector</th>
<th>shMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector A</td>
<td></td>
</tr>
<tr>
<td>Vector B</td>
<td></td>
</tr>
<tr>
<td>Vector C</td>
<td></td>
</tr>
</tbody>
</table>

Virtual Address
Detecting sharing patterns

Thread

Vector A

Vector B

Vector C

shMap

Hash Function

Virtual Address
Detecting sharing patterns

Thread access to remote caches → counter ++
Frequent access = large number of counter
Detecting sharing patterns

- Large number of counter in same entry of two threads
  - two thread access same region and share data
  - two thread locate in different chip
  - those thread need to be clustered
Spatial sampling

- Virtual address space region is mapped to shMap vector using hash function
  - Region number $>>>$ shMap entry number $->$ hash collision
Spatial sampling

- Virtual address space region is mapped to shMap vector using hash function
  - Region number >>> shMap entry number -> hash collision

- shMap filter
Spatial sampling

- Virtual address space region is mapped to shMap vector using hash function
  - Region number >>> shMap entry number -> hash collision
- shMap filter

```
shMap filter

Vector A
Vector B
```

```
Hash Function

T_A

T_B

Empty

V.A
```
Spatial sampling

• Virtual address space region is mapped to shMap vector using hash function
  – Region number \( \rightarrow \) shMap entry number \( \rightarrow \) hash collision

• shMap filter

\[
\begin{align*}
\text{V.A} & \\
\text{Hash Function} & \\
\text{V.A} & \\
\text{Vector A} & \\
\text{Vector B} & \\
\end{align*}
\]

\[
\begin{align*}
\text{shMap filter} & \\
\text{Block ID} & \\
\end{align*}
\]

\[
\begin{align*}
\hat{T}_A & \\
\hat{T}_B & \\
\end{align*}
\]
Spatial sampling

• Virtual address space region is mapped to shMap vector using hash function
  – Region number \(\gg\gg\gg\) shMap entry number \(\rightarrow\) hash collision

• shMap filter

```
shMap filter
```

```
Vector A
Vector B
```

```
Match Block ID
```

```
Hash Function
```

\(T_A\) \(T_B\)
Spatial sampling

- Virtual address space region is mapped to shMap vector using hash function
  - Region number $\rightarrow$ shMap entry number $\rightarrow$ hash collision

- shMap filter

![Diagram showing Spatial sampling process]

**Legend:**
- $T_A$
- $T_B$
- Mismatch Block ID
- Discarded
- Hash Function
- Vector A
- Vector B
- V.A
Thread similarity

- Calculate similarity($T_1, T_2$) = $\sum_{i=0}^{N} T_1[i] \cdot T_2[i]$
- Similarity($T_1, T_2$) > threshold threads $\rightarrow T_1, T_2$ in same cluster
Forming clusters

- Starts with no cluster exist
- Each thread check similarity with representative of cluster
Forming clusters

- Starts with no cluster exist
- Each thread checks similarity with representative of cluster
- No similar cluster exist → become new representative of cluster
Forming clusters

- Starts with no cluster exist
- Each thread check similarity with representative of cluster
- No similar cluster exist → become new representative of cluster
Forming clusters

• Starts with no cluster exist
• Each thread check similarity with representative of cluster
• No similar cluster exist $\rightarrow$ become new representative of cluster
Forming clusters

- Starts with no cluster exist
- Each thread check similarity with representative of cluster
- No similar cluster exist → become new representative of cluster
Forming clusters

- Starts with no cluster exist
- Each thread check similarity with representative of cluster
- No similar cluster exist → become new representative of cluster
- Similar cluster exist → participate to the cluster
Forming clusters

• Starts with no cluster exist
• Each thread check similarity with representative of cluster
• No similar cluster exist → become new representative of cluster
• Similar cluster exist → participate to the cluster
Forming clusters

- Starts with no cluster exist
- Each thread checks similarity with representative of the cluster
- No similar cluster exist → become new representative of the cluster
- Similar cluster exist → participate to the cluster
Thread migration

• Load balancing should be maintained
• Sort clusters from the largest size to the smallest size
• Assign current largest cluster to a chip with the lowest number of threads
• If necessary, cluster of threads are divided and assigned
Evaluation

- Platform specification
  - 8-way IBM Power5 machine (2x2x2 SMPxCMPxSMT)
  - 2MB L2 cache, 36MB L3 cache, 8GB RAM

- Operating System
  - Based linux-2.6.15
  - modify CPU scheduler code

- Workload
  - VolanoMark, Internet chat server
  - SPECjbb2000, Java-based server workload
  - RUBiS, database workload
Evaluation

- Reduction of remote stalls and speed up
- Compared with linux-2.6.15
Limitation

• Thread clustering cannot guarantee priority and fairness
  – If big cluster is consist of low priority tasks, that tasks occupy one chip.

• Thread clustering can cause local cache contention problem
  – The local cache may not have sufficient capacity to contain the aggregate working set of the threads.
Conclusion

• Sharing-aware scheduling is efficiently responds to newly appeared performance problem with system changing

• It can reduce remote cache access by migrating heavy-sharing threads onto same chip.

• But, it ignore priority and fairness
Question?