DoPipe: An Effective Approach to Parallelize Simulation

Parallel Processing Institute
Fudan University
2008
Outline

- Motivation
- Design Issues
- Experiments
- Work ongoing
Motivation

• General Flow of Simulation

```c
for(; ;) {
  stageN();
  stageN-1();
  ...
  stage2();
  stage1();
}
```

How to effectively utilize the increasingly popular multicore processors to accelerate the simulation process?

Running on a single-core processor
Motivation

- Parallelize the simulation by pipelining the simulated stages

For running on a single-core processor:

```
for(; ;)
{
    stageN();
    stageN-1();
    ...
    stage2();
    stage1();
}
```

For running on a dual-core processor:

```
Core 0
for(; ;)
{
    stageN();
    ...
    stageM();
}
```
```
Core 1
for(; ;)
{
    stageM-1();
    ...
    stage1();
}
```
Motivation

• Traditional dopipe
  – Back-edge dependency
    • Lower the opportunity of partition
      – Exasperate the workload imbalance
  – Communication
    • Depend on hardware support
      – Low access latency synchronization array
Design Issues

• Speculation
• Communication & Synchronization
Speculation

• To keep the computing core as busy as possible
  – Simulate the next cycle as long as the correctness is not violated
  – Temporally save the calculated result into a buffer and wait for being consumed by other core(s)
  – Stop the progress when the buffer is full

Speedup percentage will be severely limited when the computing cores are synchronized cycle by cycle.
Communication & Synchronization

• Notify the other core(s) immediately after the data is completely produced
  – Reduce the other cores’ waiting time.

• Access the data produced by the other core(s) as late as possible
  – Reduce the current core’s waiting time.

A step to further improve the performance.
Implementation

• Stage cutting

DoPipe: A Effective Approach to Parallelize Simulation
Communication

• Candidates available in Linux
  – Message
  – Pipe
  – FIFO
  – Shared Memory ✓
  – …

DoPipe: A Effective Approach to Parallelize Simulation
Communication (cont.)

- Example: 2-core dopipe
  - Producer/Consumer relationship

```c
for(; ;)
{
  stageN();
  ...
  stageM();
}
```

```c
for(; ;)
{
  stageM-1();
  ...
  stage1();
}
```
Synchronization

• Candidates available in Linux
  – Lock, Mutex, Semaphore...

• Our choice
  – Test-and-set shared variable
  – Example: 2-core dopipe

```c
load buf0_status;
while (buf0_status!=OLD)
  load buf0_status;
write buf0

load buf0_status;
while (buf0_status!=NEW)
  load buf0_status;
read buf0
```

Core 0  Core 1
Experiment Platform

• Simulator
  – SimpleScalar3.0d
  – Config: shared & private L2 cache

• Benchmark
  – gcc,gzip,art
  – Input: ref
Hardware Platforms

- Two different Intel processors

<table>
<thead>
<tr>
<th></th>
<th>E6300</th>
<th>Q6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tech.</td>
<td>65nm</td>
<td>65nm</td>
</tr>
<tr>
<td>#cores</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.86GHz</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>Trace Cache</td>
<td>2 x 32KB</td>
<td>4 x 32KB</td>
</tr>
<tr>
<td>Data Cache</td>
<td>2 x 32KB</td>
<td>4 x 32KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2MB</td>
<td>2 x 4MB</td>
</tr>
<tr>
<td>Front Side Bus</td>
<td>1066.7 MHz</td>
<td>1066 MHz</td>
</tr>
<tr>
<td>Memory Frequency</td>
<td>333.3 MHz</td>
<td>400 MHz</td>
</tr>
<tr>
<td>Memory Size</td>
<td>2GB</td>
<td>4GB</td>
</tr>
</tbody>
</table>
Experimental Results

- Speedup on 4-core Q6600
Private L2 vs Shared L2

• Private L2
  – More aggressively optimization can be done since il2 and dl2 are exclusively accessed by different cores

• Shared L2
  – Careful checks must be performed before any speculation.

So, Dopipe should theoretically reach relatively higher speedup under private L2 cache configuration.
Experimental Results

- Speedup Percentage (Shared L2 Cache)

<table>
<thead>
<tr>
<th>Application</th>
<th>E6300</th>
<th>Q6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>20.74%</td>
<td>62.83%</td>
</tr>
<tr>
<td>gcc</td>
<td>21.06%</td>
<td>64.25%</td>
</tr>
<tr>
<td>art</td>
<td>11.00%</td>
<td>76.23%</td>
</tr>
<tr>
<td>Average</td>
<td>16.87%</td>
<td>67.51%</td>
</tr>
</tbody>
</table>
Experimental Results

- Speedup Percentage (Private L2 Cache)
Reasons for Speedup Gap

• Q6600 over E6300
  – More powerful computation capability
  – Larger L2 Cache size
  – Faster inter-core communication
  – Faster memory access

To Conclude, Q6600 saves much less time on waiting for shared data, which means much better CPU utilization (also higher speedup)!

Critical factors since our dopipe relies on shared memory to implement communication and synchronization.
Work Ongoing

• Develop a 4-core version of dopipe SimpleScalar