Architectural Techniques for Improving Fine-grain Multiprocessor Performance

Brian Grayson

http://www.ece.utexas.edu/projects/armadillo/

Department of Electrical and Computer Engineering
The University of Texas at Austin

Feb 22, 1999
Shared-memory Architecture

- processor
- one or more levels of cache

interconnect

- processor
- one or more levels of cache
Our objective is to develop techniques that expose as much parallelism as possible around expensive operations.

What are the fundamental limits of parallelism around these operations, especially for fine-grain programs?
Motivation

Our goal is not to improve performance for coarse-grained applications, where the costs of “synchronization” or “communication” can be easily amortized.

Rather, we are attempting to reduce the effective cost of the expensive operations to the near-minimum, so that fine-grain applications will become more efficient.
Outline

- Introduction
- Motivation
- Background
- Shared-memory
- Message-passing
- Results and analysis
- Conclusion
Modern processors detect and maintain read-after-write dependences (RAW, data). WAR and WAW dependences are removed through register renaming.

Original code:

```assembly
add r3, r1, r2  # r3 = r1 + r2
store r3, r4    # store r3 to address r4
load r2, r5     # load r2 from address r5
add r3, r2, 4   # r3 = r2 + 4
```

After register renaming:

```assembly
add r7, r6, 4   # r7 = r6 + 4
```

Brian Grayson — Feb 22, 1999
Background: Memory Dependences

Modern processors also detect and maintain RAW, WAR, and WAW dependences for memory locations.

```
store r0, 2000
  . . .
load  r1, 2000
  . . .
store r2, 2000
  . . .
load  r3, r6
  . . .
store r5, 2000
```

Unfortunately, “memory disambiguation” makes this more difficult than tracking register dependences.
Background: Control Dependences

Branch instructions introduce “control dependences.”

```
loadimm r2, 0
loop:
  load   r1, r1+4
  add    r2, r2, 1
  compare r1, 0
  bne    loop
  store  r2, 2000

...  
```

Branch prediction: “guess” whether the branch will be taken, and start executing instructions from there.

However, branch predictions are not always correct.
Techniques for Tolerating Latency

- Reduce the latency: technology, caching.

- Find independent work: pipelining, OOO execution, multithreading.
Techniques for Tolerating Latency (continued)

- Reduce amount of dependent work: speculative execution, prediction, memory model, architecture changes.

![Diagram showing original code, out-of-order execution, and dependence change](image-url)
Dependence Conversion

Modify a conventional architecture to expose more instruction-level parallelism around thread-level synchronization operations:

- Remove unnecessary dependences.
- Weaken overly-restrictive dependences.
- Convert “expensive” dependences into more-efficient dependences.

Such changes allow the processor to use existing ILP techniques (OOO execution, etc.) to exploit this new-found parallelism.
Example of a Critical Section

. . .
do
    isSuccess = try_acquire(p->lock);
    while (not isSuccess);
memory barrier
store 1 @ p->val
load r2 @ done_counter
add r3, r2, 1
store r3 @ done_counter
memory barrier
store 0 @ p->lock  /* release(p->lock) */
. . .

Two impediments to more ILP: spin-loop and memory barriers.
Blocking Acquires

Goal: Remove the hard-to-predict control dependence in the acquire.

Key observation: “no you don’t have the lock, no, no, no, no, ..., yes you have the lock” is virtually indistinguishable from a very slow “yes you have the lock now.”

Replace the entire spin-loop acquire with a single blocking acquire instruction: the instruction completes only when the lock has been acquired.
Example with Blocking Acquire

...  

**blocking-acquire for p->lock**
memory barrier
store 1 @ p->val
load r2 @ done_counter
add r3, r2, 1
store r3 @ done_counter
memory barrier
store 0 @ p->lock   /* release(p->lock) */

...
Synthetic Dependences

Goal: Weaken the first memory barrier by explicitly adding the exact dependences.

Synthetic dependence: any dependence artificially added by the programmer to enforce the execution ordering between any two otherwise-unordered instructions.

loaddep: wait for $r_{\text{syndep-in}}$, $r_{\text{data}} = M[r_{\text{addr}}]$ and $r_{\text{syndep-out}} = 0$.

storedep: wait for $r_{\text{syndep-in}}$, $M[r_{\text{addr}}] = r_{\text{data}}$ and $r_{\text{syndep-out}} = 0$. 
Example with Synthetic Dependences

... blocking-acquire for p->lock, update r1
store dep wait on r1, 1 @ p->val
load dep wait on r1, r2 @ done_counter
add r3, r2, 1
store dep wait on r1, r3 @ done_counter
memory barrier
store 0 @ p->lock      /* release(p->lock) */
...
Partial Barriers/CSBegin & CSEnd

Using only syndeps is not scalable, and requires modification of all critical section code.

Goal: Provide some form of partial barriers.

Use CSBegin and CSEnd to “bracket” the critical section. All memory accesses in the critical section are marked by the decoder. Marked accesses stall until after CSBegin completes, CSEnd stalls until all marked accesses complete.
Example with CSBegin & CSEnd

...  
blocking-acquire @ p->lock, update r1
CSBegin wait on r1
store 1 @ p->val
load r2 @ done_counter
add r3, r2, 1
store r3 @ done_counter
CSEnd, update r11
storedep wait on r11, 0 @ p->lock
  ...  

Brian Grayson – Feb 22, 1999
Execution graph

**Traditional code:**

```
普通代码
```

```
acq   sync   CS   sync   rel
普通代码
```

**New code:**

```
普通代码
```

```
acq   CS   rel
普通代码
```

Brian Grayson – Feb 22, 1999
do
   gotANewMesg = HasMessageArrived();
   while (gotANewMesg == false);
memory barrier
access message buffer
   . . .
memory barrier
deallocate message buffer
   . . .
Message-passing Example, with blocking load, syndeps, and CSBegin/CSEnd

... blocking-load for msg-arrived, updating r1
CSBegin wait on r1
access message buffer
... CSEnd, updating r2
deallocate message buffer, wait on r2
Research Platform: Armadillo

Armadillo is a portable, executable-driven simulator that models a modern multiprocessor system.

PowerPC executable

input files for simulated executable

Armadillo, running on the host machine (x86, UltraSparc, Alpha, PowerPC)

output from simulated executable

statistics, etc.
Simulation results: Shared-memory

Microbenchmark: histogram

Results for 16 processors, 32 instruction-window, 10:1 bus

Execution time (in thousands of cycles)

Number of shared counters (m)
Simulation results: Shared-memory

4-processor SMP with 10x bus, around 10M to 50M insns

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base runtime (TTS &amp; MB)</th>
<th>Savings with all techniques</th>
<th>Number of critical sections</th>
<th>Savings per critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort 100K</td>
<td>9048968</td>
<td>68956 (1.01)</td>
<td>1580</td>
<td>43.64</td>
</tr>
<tr>
<td>barnes 512</td>
<td>11805934</td>
<td>52596 (1.00)</td>
<td>2232</td>
<td>23.56</td>
</tr>
<tr>
<td>ocean 34</td>
<td>16696549</td>
<td>42583 (1.00)</td>
<td>1532</td>
<td>27.80</td>
</tr>
<tr>
<td>raytrace checksmall</td>
<td>11417485</td>
<td>903087 (1.09)</td>
<td>20625</td>
<td>43.79</td>
</tr>
</tbody>
</table>
Message-passing Virtual Architecture

- PowerPC processor
- one or more levels of cache
- Main memory (DRAM)
- Network Interface (NI) to the network
Simulation results: Message-passing

MVP+ libraries: message-passing libraries, tuned for our network interface.

- MVP+: uses spin-loop polling and memory barriers.
- MVP+ & BL: replaces spin-loop poll with blocking load.
- MVP+ & BL & lsdep: also replaces some barriers with syndeps.
- MVP+ & BL & lsdep & PB: replace remaining barriers with partial barriers.

Benchmarks: ping (with add), mmult, SOR, et al.
Execution time versus net latency for ping, 64-entry IW

- MVP+ with poll
- MVP+ & BL
- MVP+ & BL & LSDEP
- MVP+ & BL & LSDEP & CSBegin/CSEnd
Execution time versus net latency for mmult, 64-entry IW

Execution time per iteration

Network Latency in cycles

MVP+ with poll
MVP+ & BL
MVP+ & BL & LSDEP
MVP+ & BL & LSDEP & CSBegin/CSEnd
Execution time versus net latency for mmult, with varying IW sizes

- MVP+ with poll
- MVP+ & all, 32-entry IW
- MVP+ & all, 64-entry IW
- MVP+ & all, 128-entry IW
- MVP+ & all, 256-entry IW
- MVP+ & all, 512-entry IW
- MVP+ & all, 1024-entry IW
Execution time versus net latency for 16x16 SOR, 64-entry IW

- MVP+ with poll
- MVP+ & BL
- MVP+ & BL & LSDEP
- MVP+ & BL & LSDEP & CSBegin/CSEnd

Network Latency in cycles

Execution time per iteration
Execution time versus net latency for 16x16 SOR, with varying IW sizes

- MVP+ with poll
- MVP+ & all, 32-entry IW
- MVP+ & all, 64-entry IW
- MVP+ & all, 128-entry IW
- MVP+ & all, 256-entry IW
- MVP+ & all, 512-entry IW
- MVP+ & all, 1024-entry IW
Execution time versus net latency for 32x32 SOR, with varying IW sizes

- MVP+ with poll
- MVP+ & all, 32-entry IW
- MVP+ & all, 64-entry IW
- MVP+ & all, 128-entry IW
- MVP+ & all, 256-entry IW
- MVP+ & all, 512-entry IW
- MVP+ & all, 1024-entry IW
Conclusion

- Synchronization (e.g., device or critical section) is not inherently “ILP-unfriendly.”

- New techniques (blocking-acquires/blocking-loads, synthetic dependences, partial barriers) can expose more ILP to the processor by providing precise light-weight dependences.

- These techniques are orthogonal to many other techniques, so benefits are frequently additive.
Conclusion

- The new techniques require only simple changes to current commodity microprocessors.

- Uses of these techniques include device or coprocessor interaction, even on uniprocessors.

bgrayson@ece.utexas.edu
http://www.ece.utexas.edu/projects/armadillo/