Recitation 5

Cell Profiling Tools
Agenda

● Cell Simulator Overview
● Dynamic Profiling Using Counters
● Instruction Scheduling
Cell Simulator Highlights

- Full system simulator can help in debugging and performance optimization
  - Uni-Cell and multi-Cell simulation
  - GUI user Interfaces
  - Cycle accurate SPU simulation
  - Facility for tracing and viewing simulation events

- Note: does not accurately model communication cost
Run Cell Simulator

- **Launch simulator GUI interface**
  
  ```bash
  % export SYSTEMSIM_TOP=/opt/ibm/systemsim-cell
  /opt/ibm/systemsim-cell/bin/systemsim -g &
  ```

  - Then click "go"
Main GUI Interface
Simulated Linux Environments

- Simulated Linux shell as if running on Cell hardware
Simulated and Native Linux Interoperability

- Simulated Linux has its own file system
- Files can be transferred between the native file system and the simulated file system using the `callthru` utility
- Example: transfer and execute a Cell program

```
% callthru /tmp/hello-world > hello-world
% chmod u+x hello-world
% ./hello-world
```
### Debugging

- **View machine state**

![Image of window showing machine state](image)

**REG0**
- 0x00000000000000000000000000000000

**REG1**
- 0x000003f5000000000000000000000000

**REG2**
- 0x00000000000000000000000000000000

**REG3**
- 0x00000000000000000000000000000000

**REG4**
- 0x00000000000000000000000000000000

**REG5**
- 0x00000000000000000000000000000000

**REG6**
- 0x00000000000000000000000000000000

**REG7**
- 0x00000000000000000000000000000000

**REG8**
- 0x00000000000000000000000000000000

**REG9**
- 0x00000000000000000000000000000000

**REG10**
- 0x00000000000000000000000000000000

**REG11**
- 0x00000000000000000000000000000000

**REG12**
- 0x00000000000000000000000000000000

**REG13**
- 0x00000000000000000000000000000000

**REG14**
- 0x00000000000000000000000000000000

**REG15**
- 0x00000000000000000000000000000000

**FPSCR**
- 0x00000000000000000000000000000000

**Status**
- 0x3ffeb0002 \{ stopped stop&signal \} \{ not stalled \}
Profiling

- Dynamic profiling and statistics
  - Separate stats for PPU and each SPU
Code Instrumentation and Profiling

- Fine-grained measurements during simulation are possible via `prof_*` routines
  - Profiling routines are no-ops on the Cell hardware

```c
#include <profile.h>

...  
prof_clear();
prof_start();
    function_of_interest();
prof_stop();
```
Cell Simulator Availability

- Simulator is not installed on the PS3 hardware
- Contact TAs if you want to run the simulator
Agenda

- Cell Simulator Overview
- Dynamic Profiling Using Counters
- Instruction Scheduling
Performance Counters on the SPUs

● Each SPU has a counter that counts down at a fixed rate (decrementer)
  ■ Can be used as a clock
  ■ Suitable for coarse-grained timing (1000s of instructions)
Decrementer Example

#define DECR_MAX 0xFFFFFFFF
#define DECR_COUNT DECR_MAX

// Start counting
spu_writech(SPU_WrDec, DECR_COUNT);
spu_writech(SPU_WrEventMask, MFC_DECURRENTER_EVENT);
start = spu_readch(SPU_RdDec);
    function_of_interest();

// Stop counting, print count
end = spu_readch(SPU_RdDec);
printf("Time elapsed: %d\n", start - end);
spu_writech(SPU_WrEventMask, 0);
spu_writech(SPU_WrEventAck, MFC_DECURRENTER_EVENT);
Agenda

- Cell Simulator Overview
- Dynamic Profiling Using Counters
- Instruction Scheduling
Review: Instruction Scheduling

- Instructions mostly of the form
  \[ r_3 = f(r_1, r_2) \]
  - Assembly file is a human-readable representation of these instructions
- Conceptually, instructions execute in the order in which they appear in assembly
Review: Instruction Scheduling

- With pipelining, order of instructions is important!
  - Pipeline stalls while waiting for dependencies to complete

execute b before a

a:ADD r3,r1,r2
b:ADD r6,r4,r5
c:ADD r8,r6,r7

• c flow dependent on b
• Assume 2 cycles operation latency
Static Profiling

- Use static profiling to see where stalls happen
- Generate assembly and instruction schedule
  - Manually
    - `# generate assembly (xlc -S also works)`
    - `% gcc -S filename.c`
    - `# generate timing information`
    - `% /opt/ibm/cell-sdk/prototype/bin/spu_timing -running-count ./filename.s`
      - Output stored in `filename.s.timing`
      - `-running-count` shows cycles elapsed after each instruction
  - With our Makefile
    - `% SPU_TIMING=1 make filename.s`
Reading the Assembly

- Instructions of the form
  \texttt{OP DEST SRC1 SRC2 ...}

- Header indicates source files:
  \texttt{.file "dist_spu.c"}
  \texttt{.file 1 "dist_spu.c"}
  \texttt{.file 2 "/opt/ibmcmp/xlc/8.1/include/spu_intrinsics.h"}

- Markers for source lines:
  \texttt{.LS_p1_f1_l19:}
  \texttt{.loc 1 19 0}
  \texttt{ila $7,a}
Interpreting Static Profiler Output

<table>
<thead>
<tr>
<th>Pipeline No.</th>
<th>One digit for each cycle</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>129 0D 90</strong></td>
<td>ai $6,$6,-1</td>
<td></td>
</tr>
<tr>
<td><strong>129 1D 9012</strong></td>
<td>cwx $12,$5,$2</td>
<td></td>
</tr>
<tr>
<td><strong>133 1 ---3456</strong></td>
<td>rotqby $8,$8,$10</td>
<td></td>
</tr>
<tr>
<td><strong>134 1 4567</strong></td>
<td>fm $8,$8,$9</td>
<td></td>
</tr>
<tr>
<td><strong>138 0D ---890123</strong></td>
<td>rotqby $9,$9,$11</td>
<td></td>
</tr>
<tr>
<td><strong>138 1D 890123</strong></td>
<td>lqx $9,$5,$2</td>
<td></td>
</tr>
<tr>
<td><strong>144 1 -----4567</strong></td>
<td>shufb $8,$8,$9,$12</td>
<td></td>
</tr>
</tbody>
</table>

D for dual-issue

- for stalls

-running-count adds cycle count column

opcode

129 - 128

Example: `rotqby` requires 4 cycles to complete
Instruction Scheduling on Cell

- In-order execution
- Dual pipeline
  - Pipeline selected based on instruction type
  - Two instructions can be issued simultaneously when dependencies allow
- Goal: scheduling instructions to minimize stalls
  - Loads, fp instructions liable to take a long time
  - Dual-issue whenever possible
  - IPC = 2 (instructions per cycles)
  - CPI = .5 (cycles per instruction)
Example Schedule Optimization

(dist_spu.s line 246) .LS_p1_f1_l126:
  .loc 1 26 0
  or $2,$3,$3
  ila $3,dist
  lqd $4,80($1)
  shli $4,$4,8
  lqd $5,96($1)
  shli $5,$5,2
  ---67 a $4,$4,$5
Example Schedule Optimization

(dist_spus line 246) .LS_p1_f1_l126:
.loc 1 26 0

789012    lqd     $4,80 ($1)
890123    lqd     $5,96 ($1)
  90      or      $2,$3,$3
   01      ila     $3,dist
  --3456   shli    $4,$4,8
          shli    $5,$5,2
   ---89   a       $4,$4,$5

8 cycles saved
Exercise 1 (10 minutes)

- Improve performance by rescheduling instructions
  - `tar zxf rec5.tar.gz`
  - `cd rec5/lab1/spu`

- Examine assembly code
  - `export CELL_TOP=/opt/ibm/cell-sdk/prototype`
  - `SPU_TIMING=1 make dist_spu.s`
  - Find an opportunity for performance gain via instruction scheduling and implement it (e.g., reduce stalls after `lqd` instructions near line 246)

- Generate object file from assembly
  - `./make-obj-file; cd ..; make`
  - `make-obj-file` compiles your modified assembly to binary, otherwise your optimization is lost

- Run and evaluate
  - How many cycles did you save?
    - `/opt/ibm/cell-sdk/bin/spu_timing -running-count dist_spu.s`
  - Is the new code correct?
    - Run and check if correctness test passes
Instruction Scheduling

- Compilers are very good at doing this automatically
  - Unoptimized code: 469 cycles
  - Optimized code (xlc -O5): 188 cycles
- Hand-reordering of optimized assembly is unlikely to produce significant gains except in extreme scenarios
Notes on Static Profiling

- Static profiler presents a skewed view of conditionals, loops
  - 8 cycles saved in the static schedule → how many cycles saved when the program runs?
- Data-dependent behavior not captured
  - Static profiler does not factor in loop trip counts or branch frequencies
  - Profiling doesn't account for branch misprediction
Improving Branch Prediction

- Static branch hinting from source code
  - `if(__builtin_expect(CONDITION, EXPECTED))`
  - Useful macros:
    - `#define LIKELY(exp)   __builtin_expect(exp, TRUE)`
    - `#define UNLIKELY(exp) __builtin_expect(exp, FALSE)`
    - `if(LIKELY(i == j)) { ... }`
Summary

- Static and dynamic profiling tools are used to identify performance bottlenecks

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cell simulator</strong></td>
<td>Good statistics on stall sources; no recompile needed</td>
<td>Simulator is slow</td>
</tr>
<tr>
<td>Use to get statistical info on program runs</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Decrementers</strong></td>
<td>Easy to set up</td>
<td>Little insight into sources of stalls</td>
</tr>
<tr>
<td>Use to measure runtime for a segment of code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Schedule analysis</strong></td>
<td>Identifies exactly where time is spent</td>
<td>Low level; only does straight-line analysis</td>
</tr>
<tr>
<td>Use to see instruction-level interactions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>