

MIT OpenCourseWare

<http://ocw.mit.edu>

6.189 Multicore Programming Primer, January (IAP) 2007

Please use the following citation format:

Phil Sung, *6.189 Multicore Programming Primer, January (IAP) 2007*.
(Massachusetts Institute of Technology: MIT OpenCourseWare).
<http://ocw.mit.edu> (accessed MM DD, YYYY). License: Creative
Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:

<http://ocw.mit.edu/terms>

6.189 IAP 2007

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

```
% 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

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

```
#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_writetech(SPU_WrDec, DECR_COUNT);
spu_writetech(SPU_WrEventMask, MFC_DECREMENTER_EVENT);
start = spu_readch(SPU_RdDec);

    function_of_interest();

// Stop counting, print count
end = spu_readch(SPU_RdDec);

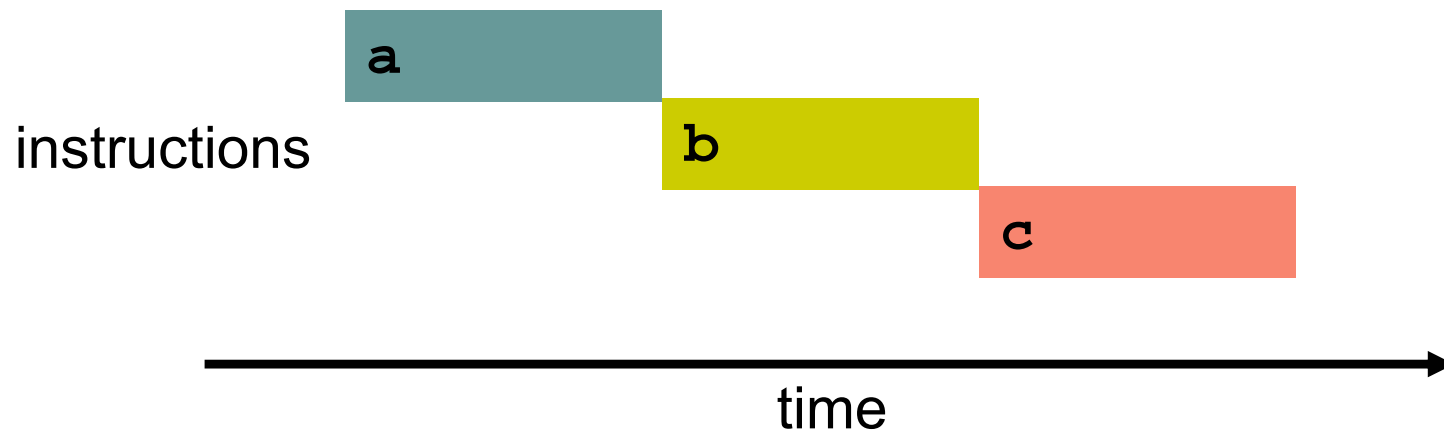
printf("Time elapsed: %d\n", start - end);
spu_writetech(SPU_WrEventMask, 0);
spu_writetech(SPU_WrEventAck, MFC_DECREMENTER_EVENT);
```

Agenda

- Cell Simulator Overview
- Dynamic Profiling Using Counters
- **Instruction Scheduling**

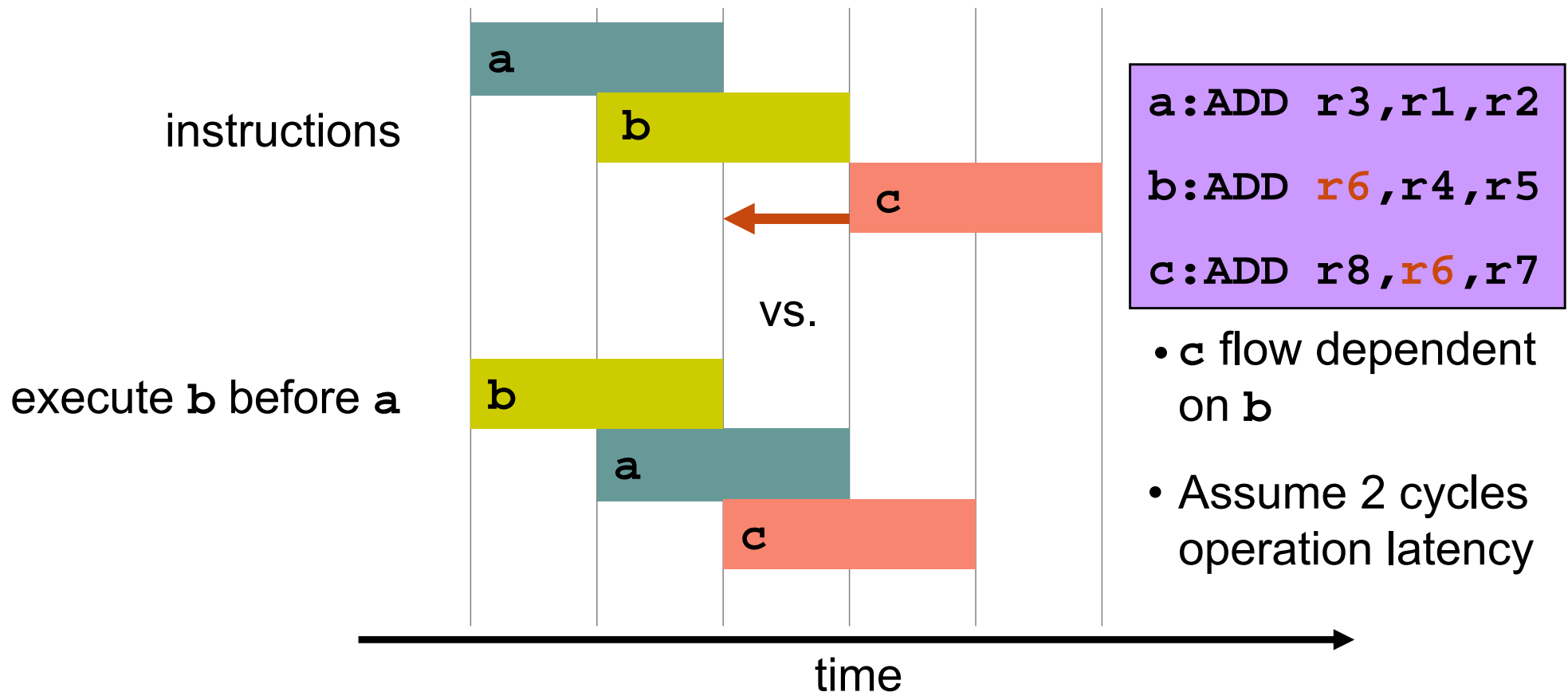
Review: Instruction Scheduling

- Instructions mostly of the form
 $r3 = f(r1, r2)$
 - 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



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
```

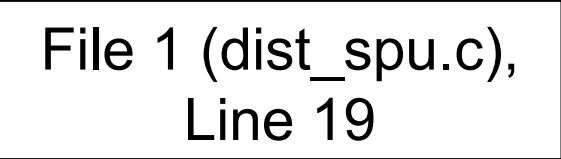
Reading the Assembly

- Instructions of the form
`OP DEST SRC1 SRC2 ...`
- Header indicates source files:

```
.file "dist_spu.c"  
.file 1 "dist_spu.c"  
.file 2 "/opt/ibmcmp/xlc/8.1/include/spu_intrinsics.h"
```

- Markers for source lines:

```
.LS_p1_f1_l19:  
.loc 1 19 0  
ila          $7,a
```



Interpreting Static Profiler Output

	Pipeline No.			Assembly
		One digit for each cycle (example: rotqby requires 4 cycles to complete)		Opcode Operands
129	0D	90		ai \$6, \$6, -1
129	1D	9012		cwx \$12, \$5, \$2
133	1	---3456		rotqby \$8, \$8, \$10
134	1	4567	- for stalls	rotqby \$9, \$9, \$11
138	0D	---890123		fm \$8, \$8, \$9
138	1D	890123		lqx \$9, \$5, \$2
144	1	-----4567		shufb \$8, \$8, \$9, \$12

D for dual-issue

-running-count adds cycle count column

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_126:
                        .loc 1 26 0
78                      or    $2,$3,$3
                        89     ila  $3,dist
                        901234  lqd  $4,80($1)
                        -----5678  shli $4,$4,8
                        678901  lqd  $5,96($1)
                        -----2345  shli $5,$5,2
                        ---67 a    $4,$4,$5
```

Example Schedule Optimization

```
(dist_spu.s line 246) .LS_p1_f1_l26:
                        .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
4567                  shli   $5,$5,2
---89                 a      $4,$4,$5
```

8 cycles saved

Exercise 1 (10 minutes)

- Improve performance by rescheduling instructions
 - See example code in recitations section.
 - `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 (`x1c -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

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