Lecture 13: Things of Interest

G63.2011.002/G22.2945.001 · November 30, 2010
Outline

Debugging

Instrumentation

Profiling and Hardware
“Odds and Ends”

- Tools (emphasis on Linux, non-proprietary)
- Ways to use them
- Learn new details about hardware along the way
- ... across our four ways of high-performance computing: Serial, OpenMP, MPI, GPU

Will post slides, video (hopefully)

Questions about your final project?

→ Ask us! We’re happy to help!
Outline

Debugging

Instrumentation

Profiling and Hardware
Debugging

Bad program behavior:
• Wrong result
• Segmentation fault
• Run-time errors
• assert() violations (<assert.h>, -DNDEBUG)

Desired Insight:
• Where? (Source code location)
• When? (Execution History)
  • History within function
  • Call stack
• With what data? (Variable contents, etc.)
• → Why? (And how do I fix it?)

Key Actions: Attach to inferior, trace (ptrace()) its execution
Debugging

Bad program behavior:
- Wrong result
- Segmentation fault
- Run-time errors
- `assert()` violations (`<assert.h>`, `-DNDEBUG`)

Desired Insight:
- Where? (Source code location)
- When? (Execution History)
  - History within function
  - Call stack
- With what data? (Variable contents, etc.)

What about bugs that aren’t reproducible?

Key Actions: Attach to inferior, trace (ptrace()) its execution
Debugging with GDB: Summary

• Three main usage patterns:
  • Run-until-crash (‘Post-mortem’)
  • Core dump
  • Break-and-trace

• \(-g\) vs \(-O\)

• Ctrl X, Ctrl A

• Step into (s), step over (n), finish (fin)

• p data to look at variables
Other Debuggers: DDD

GNU Data Display Debugger (Free)
Other Debuggers: TotalView

TotalView (Proprietary)
Other Debuggers: DDT

Allinea Distributed Debugging Tool (Proprietary)
Outline

Debugging

Instrumentation

Profiling and Hardware
Problem: Debugging only deals with problems when they cause observable wrong behavior (e.g. a crash).

Doesn’t find latent problems.

Suggested solution: Monitor program behavior (precisely) while it’s executing. Possible?
What is Instrumentation?
A.k.a. how does Valgrind work?

Tools:
- Memcheck (find pointer bugs)
- Massif (find memory allocations)
- Cachegrind/Callgrind (find cache misbehavior)
- Helgrind/DRD (find data races)
Outline

Debugging

Instrumentation

Profiling and Hardware
Profilers

Slow program execution:
- Poor memory access pattern
- Expensive processing (e.g. division, transcendental functions)
- Control overhead (branches, function calls)

Desired Insight:
- Where is time spent? (Source code location)
- When? (Execution History)
  - Call stack
- What is the limiting factor?

Main Types of Profilers:
- Exact, Sampling
- Hardware, Software
Reflections on Profilers

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Fast</td>
<td>- Slow</td>
</tr>
<tr>
<td>- Noisy</td>
<td>+ Exact</td>
</tr>
<tr>
<td>(takes time to converge!)</td>
<td></td>
</tr>
</tbody>
</table>

No free lunch. But: No exact machine-level profiler!
What do OProfile sample counts mean?

Individually: not much!

→ Ratios make sense!

What kind of ratios?

• \((\text{Events in Routine 1})/(\text{Events in Routine 2})\)
• \((\text{Events in Line 1})/(\text{Events in Line 2})\)
• \((\text{Count of Event 1 in X})/(\text{Count of Event 2 in X})\)

**Always ask:** Sample count sufficiently converged?
**OProfile: Examples I**

- (DCU_LINES_IN or L1D_REPL) / INST RETIRED
  - L1 miss rate, target: small, location understood (seen)

- L2_LINES_IN / INST RETIRED
  - L2 miss rate, target: small

- INST RETIRED / CPU_CLK_UNHALTED
  - Instructions per clock, target > 1 (seen)

- CYCLES_L1I_MEM_STALLED / CPU_CLK_UNHALTED
  - Instruction fetch stalls. Should never happen—means CPU could not predict where code is going. (pipeline stall)

- BR_IND_CALL_EXEC / INST RETIRED
  - Fraction of indirect calls (virtual table lookups)
- **L1D_CACHE_LD / CPU_CLK_UNHALTED**
  Fraction of time the L1 load/store buffers are full

- **STORE_BLOCK / CPU_CLK_UNHALTED**
  Fraction of cycle CPU is blocked waiting to be able to write to memory

- **PAGE_WALKS / CPU_CLK_UNHALTED**
  Cycles spent waiting for page table walks (TLB miss penalty)

- **DTLB_MISSES / INST_RETIRED**
  Data TLB miss rate
Virtual Memory

Virtual address space

0x00000000
0x00010000
0x10000000
0x7fffffff

Physical address space

0x00000000
0x00ffffff

page belonging to process
page not belonging to process
text
data
stack
Virtual Memory

Linear address:

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>

* 32 bits aligned to a 4-KByte boundary

One page directory per process.

Debugging Instrumentation Profiling and Hardware
Virtual Memory

Linear address:

31  24  23  16  15  8  7  0

... and two extra memory accesses *per memory access?*
What leads to TLB flush?

TLB flush $\Rightarrow$ Cache flush?
Caching the Page Table

virtual address → TLB → physical address

- TLB hit
- TLB miss
- TLB write

page table

- page table hit
- page not present
- page table write

What leads to TLB flush?
TLB flush ⇒ Cache flush?
Influencing TLB performance

What to do if limited by TLB performance?

- Access fewer pages:
  - Increase locality
  - Problem: fragmented memory!

- Default x86 page granularity: 4 kiB
  “Huge” pages also exist: 2 MiB

Obtaining huge-page memory: (Linux only)

- mount -t hugetlbfs none /mnt/huge
- Create /mnt/huge/myfile
- mmap() that file.

→ 5–10% gain on matmul

But: Huge pages are shared, scarce resource!
OProfile: Also for multi-processor programs

- **EXT_SNOOP / INST_RETIRED**
  Fraction of instructions causing retrieval of modified cache line from other core

- **(L1D_CACHE_LOCK_DURATION + 20 × L1D_CACHE_LOCK)/CPU_CLK_UNHALTED**
  Fraction of cycles spent waiting for synchronized ("atomic") access to memory
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

How?

OpenCL:

atomic

{add, inc, cmpxchg, . . .}

(int *global, int value);

Debugging Instrumentation Profiling and Hardware
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

OpenCL:

```c
atomic {
    add, inc, cmpxchg, ...
}(int *global, int value);
```
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible! → Interruptible!

How?

OpenCL:

\[
\text{atomic}\{\text{add, inc, cmpxchg}, \ldots\} (\text{int } *\text{global}, \text{int value});
\]
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Atomic Global Memory Update:
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Atomic Global Memory Update:
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

How? OpenCL:
atomic_{add,inc,cmpxchg, ... } (int *global, int value);
1. $ cc -pg -omy-program my-program.c
2. $ ./my-program
   (gmon.out gets created)
3. $ gprof ./my-program

Implementation:

- Change function invocation to store call graph information on every function call
- Look at the program counter/call graph $\sim$100 times per second (coarse!)
### (Abbreviated) GProf Output

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>seconds</td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80.95</td>
<td>0.17</td>
<td>0.17</td>
<td>2560</td>
<td>0.07</td>
<td>regmul</td>
</tr>
<tr>
<td>19.05</td>
<td>0.21</td>
<td>0.04</td>
<td>40</td>
<td>1.00</td>
<td>square_dgemm</td>
</tr>
<tr>
<td>0.00</td>
<td>0.21</td>
<td>0.00</td>
<td>640</td>
<td>0.00</td>
<td>copy_to_block</td>
</tr>
<tr>
<td>0.00</td>
<td>0.21</td>
<td>0.00</td>
<td>320</td>
<td>0.00</td>
<td>blockmul</td>
</tr>
</tbody>
</table>

[[…]]

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self time</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.21</td>
<td></td>
<td>&lt;spontaneous&gt; main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.17</td>
<td>40/40</td>
<td>square_dgemm_tuned [5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.04</td>
<td>0.00</td>
<td>40/40</td>
<td>square_dgemm [6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>80/80</td>
<td>fill_random_matrix [10]</td>
</tr>
</tbody>
</table>

| [3]   | 81.0   | 0.00      | 0.17     | 320/320| form_Cblock [4]             |
|       |        | 0.17      | 0.00     | 2560/2560 | regmul [2]               |
|       |        | 0.00      | 0.00     | 640/640 | copy_to_block [7]          |

Also: Annotated source.
Intel VTune (sampling, perf counters, proprietary)
AMD Code Analyst (sampling, perf counters, proprietary)
Apple Shark (sampling, proprietary)

Debugging Instrumentation Profiling and Hardware
Profiling MPI: Jumpshot

Jumpshot MPI Profiler (exact, event-based, free)

1. Install MPE. (works on top of existing MPI)
2. $ mpecc -o my-program my-program.c
3. $ mperun -np 32 my-program

What to do about deluge of data?
Now how about GPUs? (Seen in hw3: Events)

Debugging Instrumentation Profiling and Hardware
Profiling MPI: Jumpshot

1. Install MPE. *(works on top of existing MPI)*
2. `$ mpecc -omy-program my-program.c`
3. `$ mperun -np 32 my-program`

What to do about deluge of data?
Now how about GPUs? (Seen in hw3: Events)
Nvidia GPU Profiler: Events

\textbf{gld\_request} : Number of executed global load instructions per warp in a SM

\textbf{gst\_request} : Number of executed global store instructions per warp in a SM

\textbf{divergent\_branch} : Number of unique branches that diverge

\textbf{instructions} : Instructions executed

\textbf{warp\_serialized} : Number of SIMD groups that serialize on address conflicts to local memory

And many more: see

/share/apps/cuda/toolkit/3.1/doc/Compute_Profiler_3.1.txt

(Careful: CUDA terminology)
## OpenCL ↔ CUDA: A dictionary

<table>
<thead>
<tr>
<th><strong>OpenCL</strong></th>
<th><strong>CUDA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>Grid</td>
</tr>
<tr>
<td>Work Group</td>
<td>Block</td>
</tr>
<tr>
<td>Work Item</td>
<td>Thread</td>
</tr>
<tr>
<td>__kernel</td>
<td><strong>global</strong></td>
</tr>
<tr>
<td>__global</td>
<td><strong>device</strong></td>
</tr>
<tr>
<td>__local</td>
<td><strong>shared</strong></td>
</tr>
<tr>
<td>__private</td>
<td><strong>local</strong></td>
</tr>
<tr>
<td>imagend_t</td>
<td>texture&lt;type, n, ...&gt;</td>
</tr>
<tr>
<td>barrier(LMF)</td>
<td>__syncthreads()</td>
</tr>
<tr>
<td>get_local_id(012)</td>
<td>threadIdx.xyz</td>
</tr>
<tr>
<td>get_group_id(012)</td>
<td>blockIdx.xyz</td>
</tr>
<tr>
<td>get_global_id(012)</td>
<td>– (reimplement)</td>
</tr>
</tbody>
</table>
Nvidia hardware has 16 banks. Work item access local memory in groups of 16.

OK: local variable[get local id(0)], (Single cycle)

Bad: local variable[BANK COUNT*get local id(0)], (BANK COUNT cycles)

OK: local variable[(BANK COUNT+1)*get local id(0)], (Single cycle)

OK: local variable[ODD NUMBER*get local id(0)], (Single cycle)

Bad: local variable[2*get local id(0)], (BANK COUNT/2 cycles)

OK: local variable[f(blockIdx)], (Broadcast–single cycle)
Local Memory: Banking

- **Banking in Local Memory:**
  - Nvidia hardware has 16 banks.
  - Work item access local memory in groups of 16.

- **Access Patterns:**
  - **OK:** Access to local variables with addresses that are multiples of the bank count (single cycle).
  - **Bad:** Access to local variables with addresses that are not multiples of the bank count (multiple cycles).

- **Examples:**
  - Accessing local variable \[\text{get local id(0)}\] for Bank 0, 1, 2 using addresses 0, 1, 2 respectively, which are multiples of 16 (single cycle).
  - Accessing local variable \[\text{ODD NUMBER * get local id(0)}\] for Bank 3, 4, 5 using addresses 6, 9, 12 respectively, which are multiples of 16 (single cycle).
  - Accessing local variable \[\text{2 * get local id(0)}\] for Bank 6, 7, 8 using addresses 10, 14, 18 respectively, which are multiples of 16 (multiple cycles).

- **Broadcast Pattern:**
  - Accessing local variable \[f(blockIdx)\] for Bank 9, 10, 11 using addresses 15, 18, 21 respectively, which is broadcasted (single cycle).
Nvidia hardware has 16 banks. Work item access local memory in groups of 16.
Local Memory: Banking

Nvidia hardware has 16 banks. Work item access local memory in groups of 16.

**OK:** `local_variable[get_local_id(0)]`,
(Single cycle)
Local Memory: Banking

Bad: `local_variable[BANK_COUNT*get_local_id(0)]`
(BANK_COUNT cycles)
Local Memory: Banking

Nvidia hardware has 16 banks. Work item access local memory in groups of 16.

**OK:** `local_variable[(BANK_COUNT+1)*get_local_id(0)]`
(Single cycle)
Local Memory: Banking

OK: `local_variable[ODD_NUMBER*get_local_id(0)]`
(Single cycle)
Local Memory: Banking

Bad: local_variable[2*get_local_id(0)]
(BANK_COUNT/2 cycles)
Local Memory: Banking

Nvidia hardware has 16 banks. Work item access local memory in groups of 16.

**OK:** local_variable[f(blockIdx)]
(Broadcast–single cycle)
Nvidia hardware has 16 banks.  
Work item access local memory in groups of 16.
Image Credits

- Valgrind logo: Julian Seward
- Clock: sxc.hu/cema
- Bar chart: sxc.hu/miamiamia
- Dictionary: sxc.hu/topfer