

Lecture Topics

- sample-based profiling example
- capturing more details (OProfile)

Administrivia

- mail lab #1 solution to me by tonight
- EWS request for more space submitted

- flat profile information
 - tracks number of samples found inside code for each function
 - functions listed in decreasing order of frequency (sample count)
 - fields for each function
 - % of time taken by that function (fraction of samples)
 - cumulative seconds in all functions so far
 - seconds in this function
 - # calls made to this function (tracked by mcount)
 - milliseconds spent in this function per call (average)
 - milliseconds spent in this function and descendants per call (avg.)
 - function name

- call graph profile information
 - note: accounting for recursion is tricky; see notes in output or papers
 - tracks number of samples found in function and descendants
 - listed in decreasing order of frequency
 - function names annotated with rank information
 - for each parent function
 - time in function while within this parent
 - time in function's children while within this parent
 - number of calls made from parent
 - total number of calls made non-recursively
 - for reported function:
% of time, self-time, children time, calls (recursion separately)
 - for each child function
 - time in child while within reported function
 - time in grandchildren while within reported function
 - number of calls made to child from reported function
 - total number of calls to child made non-recursively

- example from RigelSim
 - simulator for 1000-core chip
 - example uses 128 cores
 - names truncated for clarity
 - optimized -O2
- total time is 664.50 seconds

%	cum.	self		
time	seconds	seconds	calls	name
64.90	431.27	431.27	1133522621	std::map<std::string, ...
9.39	493.68	62.41	28629808	CacheModel::read_access_instr
7.48	543.40	49.72	29209531	CacheModel::read_access
7.22	591.37	47.97	8387008	Cluster::step
1.18	599.20	7.83	84413863	CoreSystem::execute

%	self	calls	name
0.00		176/1133522621	ProfileStat::init(_IO_FILE*) [124]
0.01		31224/1133522621	DRAMModel::SetDataBusBusy [108]
0.01		33026/1133522621	DRAMModel::SendCommand [96]
0.02		52069/1133522621	TileInterconnectHTree::PerCycle [18]
0.03		72078/1133522621	GlobalNetworkCrossbar::PerCycle [42]
1.51		3975785/1133522621	L2Cache::PerCycle [13]
1.99		5241880/1133522621	TileInterconnectBase::PerCycle [19]
427.69	1124116383/1133522621		Cluster::step() [2]
[3] 64.9	431.27	1133522621	std::map<std::string, ...
	0.00	178/354	std::_Rb_tree [301]
	0.00	178/191	std::_Rb_tree [303]
	0.00	89/89	std::_Rb_tree [314]

- flat profile indicates that STL map call is taking a large fraction of total time
- call graph profile indicates that single parent primarily responsible (cluster step function)

- example from RigelSim

```
// Filename: rigel-sim/src/cluster.cpp
//
// Date: 2009-02-24
// Revision: 1896
// Author: John H. Kelm
//
// This excerpt is from the main pipeline model of the Rigel core. Performance
// counters are stored in an STL map that uses C strings as keys. For every
// instruction that retires, a number of performance counters are incremented.
//
// In an older version of the code, there were only counters for cache
// accesses. Only about four counters per retired instruction were accessed.
// The overhead for using a map in the older code was < 5% of the runtime.
// The recent gprof output shows hashing calls for the STL map contributing
// to >65% of the runtime. The likely cause is the ten-fold increase in
// hash lookups for retiring instructions.

// examples of stats compilef for each instruction
profiler::stats["INSTR_INSTR_STALL_CYCLES"].inc
    (instr->stats.cycles.instr_stall);
profiler::stats["INSTR_IF_OCCUPANCY"].inc(instr->stats.cycles.fetch);
profiler::stats["INSTR_DE_OCCUPANCY"].inc(instr->stats.cycles.decode);
profiler::stats["INSTR_EX_OCCUPANCY"].inc(instr->stats.cycles.execute);
profiler::stats["INSTR_MC_OCCUPANCY"].inc(instr->stats.cycles.mem);
profiler::stats["INSTR_FP_OCCUPANCY"].inc(instr->stats.cycles.fp);
```

- these are hashes of constant strings
- Why doesn't the compile optimize them away?
 - per-string nodes are added when first used
 - compiler would have to do whole-code analysis to identify set
 - do you want nodes never executed to pre-exist?
 - does the code ever use a non-static string?
 - does a non-static string ever happen to match a static one?
(do the online and compiler hashes have to match?)

- after code fixed to use enum and array
- total time is 151.15 seconds (saved more than we might expect)
 - top call ordering is identical (except for map), as are counts
 - top call timing reduced by 17 to 58% of original (extra optimizations?)
 - savings on longer runs is about a factor of two

%	cum.	self		
time	seconds	seconds	calls	name
34.20	51.70	51.70	28629808	CacheModel::read_access_instr
24.68	89.00	37.30	29209531	CacheModel::read_access
13.42	109.28	20.28	8387008	Cluster::step
2.22	112.64	3.36	84413863	CoreSystem::execute

Capturing More Details

- only instrumented calls recorded with gprof; what about
 - library routines?
 - system calls?
 - interrupt handlers?
- OProfile tool supports system-wide data collection
 - oprofile.sourceforge.net
 - operates as a kernel daemon
 - driven by processor performance counter events
 - these are not virtualized automatically!
 - OProfile counts everything that executes
 - opens gaping security holes
 - e.g., freeze machine on demand
 - be careful about who can use it

- performance counters
 - wide range available (examples from AMD Athlon)
 - data cache accesses, misses, fills from L2/memory, writebacks
 - instruction cache fetches, misses
 - data/instruction TLB misses (two levels each)
 - misaligned data references
 - retired: instructions, operations, taken/mispredicted/all branches
 - number of interrupts, cycles spent with IF=0, interrupts pending with IF=0
 - use model
 - select event counter and choose counter value
 - use privileged instructions to inform processor
 - when reached, NMI generated (processor-specific, of course!)

- OProfile approach
 - install kernel module
 - use kernel daemon to track samples
 - minimize interference overhead
 - do not dump to disk (for example) on each NMI
 - accumulate into buffer, dump once in a while
 - high water mark used to make daemon runnable
 - use model
 - start the daemon
 - start profiling (don't profiling starting the daemon!)
 - execute your program
 - stop profiling
 - (when all done, tear down daemon)
 - allows sessions to be named (not all "gmon.out")

- OProfile options
 - location of kernel image (or none)
 - buffer size, high water mark
 - call graph depth desired
 - separation of sample data
 - based on current task information and CPU ID
 - interrupts can thus be misattributed
 - e.g., my disk block can arrive while some other task runs
 - likely for interactive code, but probably not too important
 - options
 - lump everything together (default)
 - separate results per application (shared libraries or both libraries and kernel)
 - separate results per thread
 - separate results per CPU
 - data sets expand rapidly; can limit which images to track to reduce