Energy-Aware Computing

Lecture 5: Simplescalar/Wattch howtos

Outline

- How to build, configure and run
- Understanding the results
  - in particular power related
- Simplescalar code navigation
- Wattch code navigation

How to build the executable

- Download annotated/refactored (slightly) version from `http://www.inf.ed.ac.uk/teaching/courses/eac/hl.tar.gz`
  - de-compress and untar somewhere suitable
- First time only:
  - `make config-alpha`
  - `make symlinks`
- After source code changes:
  - `make`

Configure processor

- Create default configuration file `conf`:
  - `./sim-outorder –dumpconfig conf`
- Huge number of options
  - options after `bugcompat` are for decay/leakage
  - but watch out: your added options may appear later!
- Most options are self-explanatory and have a comment line describing available options
  - Also explained in user guide

Run a benchmark

- Download a benchmark from `http://www.inf.ed.ac.uk/teaching/courses/eac/traces/`
- Open the `conf` file and change the:
  `-max:inst 0` to `-max:inst 1000000`
  `-fastfwd 0` to `-fastfwd 10000000`
- The above options are just for testing
  - you should think carefully how to configure, simulate and organize files when experimenting
- `./sim-outorder –config conf <trace> >& trace.out`

Reading simulation output

- Simulation outputs a large text file containing:
  - Some cacti output, in a weird order
    - check for error messages
  - Leakage power with detailed breakdown
  - Dynamic power per unit, per operation
    - strangely starts with some processor parameters
  - Simulation configuration options & notes
  - Number of fast-forwarding instructions
  - Lots of simulation "statistics"/counters:
    - up to `mem.ptab_miss_rate`
  - Dynamic power results
    - up to `max_cycle_power_cc3`
  - Leakage power results

Power results

- XYZ power consumption (at the beginning of the output file, from dump_power_stats()) is in Watts
  - `crossover_scaling` accounts for short-circuit power. Fixed at 20% in Wattch
- (total) XYZ_power is a measure of *energy*:
  - for each cycle that XYZ is used, the power is added up. if you multiply this with the cycle time (variable `Period` in `powerinit.c`), you'll get actual energy
- avg_XYZ_power is XYZ_power/`sim_cycles`
  - (multiplied with Period) a measure of energy / cycle
  - `sim_cycles` – total number of cycles in full simulation; does not include fast forwarding

Power "statistics"

- `rename_power` includes RAT, dependency check logic (DCL), instruction decode
- `bpred_power` includes BTB, RAS, local, global predictors, chooser
- `icache_power` includes I$ and I-TLB
- `alu_power` includes integer and FP ALUs.
- `fetch_stage_power` = icache+bpred
- `dispatch_stage_power` = rename_power
- `issue_stage_power` = alu+resultbus+dcache +dcache2+window+lsq
- `total_power` = rename_power + fetch_stage_power + issue_stage_power + regfile_power + clock_power

Clock gating

- A method for saving power when idle
  - More details in future lecture
- Wattch has 4 "modes":
  - XYZ – No conditional clocking
  - XYZ_cc1 – Simple conditional clocking
  - XYZ_cc2 – Aggressive ideal cc
  - XYZ_cc3 – Aggressive non-ideal cc

Simplescalar code navigation

- We've seen some data structures last time
- Most of the code in `sim-outorder.c`
  - a beast of ~5,500 lines of code
- You will probably need to see cache.c
  - just 1,621 lines, most for leakage/decay (TBI)
- You shouldn't need to touch any other file
  - unless you add instructions, or do other major changes to the processor

Understanding simplescalar

- Start with sim_main() at end of sim-outorder.c
  - Don't get bogged down to minor details. Grasp the basics
- Then read the main functions:
  - ruu_fetch(), ruu_dispatch(), ruu_issue(), lsq_refresh(), ruu_writeback(), ruu_commit()
- Could take a full day's work, prob. more
  - Do this; it will save multiple debugging time later

sim_main()

perform the fast-forward phase
forever do
    ruu_commit ()
    ruu_release_fu()
    ruu_writeback()
    lsq_refresh()
    ruu_issue()
    ruu_dispatch()
    ruu_fetch()

- Every iteration is a *single cycle*
- Multiple instructions are handled inside most functions in a loop
  - superscalar machine: many instructions per cycle

ruu_fetch()

- Fetch and predict a number of instructions
- It stalls on I$ misses, branch misprediction
  - Using variable `ruu_fetch_issue_delay` in `sim_main()`
- Instructions placed in `fetch_data[]` circular buffer
- I$, iTLB accesses for updating their "status" and provide latency – determines hit/miss
  - Instruction actually fetched from memory
- Branch predictor can cheat
  - instruction opcode is passed to it

ruu_dispatch()

- Pick from fetchQ and decode instruction (in order)
  - in reality it also executes them
  - uses a C switch statement and lots of macros
- Breaks loads/stores into effective address calculation (into RUU) and load/store (into LSQ)
- Register renaming and dependency checking
  Using `ruu_link_idep`, `ruu_install_odep`
- Checks if operands ready and places into readyQ
- Checks for misprediction, sets `spec_mode` keeps recovery info.

ruu_issue()

- Get next ready instruction from readyQ
- Stores complete immediately
- Loads check LSQ, access D$, dTLB
- All instructions (exc stores) try to get appropriate functional unit

  fu = res_get (fu_pool, MD_OP_class (rs→op)
- Schedule future event for completion

  eventq_queue_event(rs, sim_cycle + latency)

lsq_refresh()

- Scheduling for loads/stores
- Scan LSQ in order
  - Store with unknown address, stop scanning
  - Store with unknown data, remember address in `std_unknowns`
  - Ready load matching std_unknowns, don't issue
  - Other ready loads move to readyQ

ruu_writeback()

- Gets events from eventQ, if time is right
- If “recover instruction”, squash pipe, correct PC, set ruu_fetch_issue_delay
- Update rename table
- Broadcast result to consuming instructions
  - They may become ready; place in readyQ

ruu_commit()

- Scan RUU in order
  - If instruction not complete (writeback), finish
  - If store, get mem-port, access D$, dTLB
  - Release LSQ entry for loads/stores
  - Release RUU entry

How wattch works

- At initialisation phase (run once)
  - Calculate (`calculate_power`) and report power per unit (`dump_power_stats`)
    - stored in `power` C-structure
  - Clear cumulative power (energy) global vars
- Per simulation cycle:
  - `clear_access_stats()` at beginning of cycle
  - Calculate "power" at end of cycle in `update_access_stats()`
- Most code in power.c ~2,600 lines of code
- Lots of access counter updates in `sim-outorder.c`

Now it's your turn

- You will add a filter-cache (for data accesses only) to a processor and evaluate it
- Filter cache is a small, highly-associative 0-level cache. Level 1 D$ is only accessed when filter-cache misses
- Details in coursework #1a handout
  - worth 5% of total course marks
- I will release model answer after the deadline