A User’s and Hacker’s Guide to the SimpleScalar Architectural Research Tool Set

(for tool set release 2.0)

Todd M. Austin
taustin@ichips.intel.com
Intel MicroComputer Research Labs
January, 1997
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
A Computer Architecture Simulator Primer

• What is an architectural simulator?
  - a tool that reproduces the behavior of a computing device

• Why use a simulator?
  - leverage faster, more flexible S/W development cycle
    - permits more design space exploration
    - facilitates validation before H/W becomes available
    - level of abstraction can be throttled to design task
    - possible to increase/improve system instrumentation
A Taxonomy of Simulation Tools

- Functional
  - Trace-Driven
  - Exec-Driven
- Performance
  - Inst Schedulers
  - Cycle Timers
  - Interpreters
  - Direct Execution

- shaded tools are included in the SimpleScalar tool set
Functional vs. Performance Simulators

- **functional simulators** implement the architecture
  - the architecture is what programmer’s see
- **performance simulators** implement the microarchitecture
  - model system internals (microarchitecture)
  - often concerned with time
Execution- vs. Trace-Driven Simulation

- trace-based simulation:
  - simulator reads a “trace” of inst captured during a previous execution
  - easiest to implement, no functional component needed

- execution-driven simulation:
  - simulator “runs” the program, generating a trace on-the-fly
  - more difficult to implement, but has many advantages
  - direct-execution: instrumented program runs on host
Instruction Schedulers vs. Cycle Timers

- constraint-based instruction schedulers
  - simulator schedules instructions into execution graph based on availability of microarchitecture resources
  - instructions are handled one-at-a-time and in order
  - simpler to modify, but usually less detailed

- cycle-timer simulators
  - simulator tracks microarchitecture state for each cycle
  - many instructions may be “in flight” at any time
  - simulator state == state of the microarchitecture
  - perfect for detailed microarchitecture simulation, simulator faithfully tracks microarchitecture function
The Zen of Simulator Design

- design goals will drive which aspects are optimized
- The SimpleScalar Architectural Research Tool Set
  - optimizes performance and flexibility
  - in addition, provides portability and varied detail

<table>
<thead>
<tr>
<th>Performance</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance: speeds design cycle</td>
<td>Flexibility: maximizes design scope</td>
</tr>
<tr>
<td>Detail: minimizes risk</td>
<td>Detail: minimizes risk</td>
</tr>
</tbody>
</table>

Pick Two
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
The SimpleScalar Tool Set

- computer architecture research test bed
  - compilers, assembler, linker, libraries, and simulators
  - targeted to the virtual SimpleScalar architecture
  - hosted on most any Unix-like machine
- developed during my dissertation work at UW-Madison
  - third generation simulation system (Sohi → Franklin → Austin)
  - 2.5 years to develop this incarnation
  - first public release in July ‘96, made with Doug Burger
  - second public release in January ‘97
- freely available with source and docs from UW-Madison

http://www.cs.wisc.edu/~mscalar/simplescalar.html
SimpleScalar Tool Set Overview

- compiler chain is GNU tools ported to SimpleScalar
- Fortran codes are compiled with AT&T’s f2c
- libraries are GLIBC ported to SimpleScalar
Primary Advantages

• extensible
  □ source included for everything: compiler, libraries, simulators
  □ widely encoded, user-extensible instruction format

• portable
  □ at the host, virtual target runs on most Unix-like boxes
  □ at the target, simulators can support multiple ISA’s

• detailed
  □ execution driven simulators
  □ supports wrong path execution, control and data speculation, etc...
  □ many sample simulators included

• performance (on P6-200)
  □ Sim-Fast: 4+ MIPS
  □ Sim-OutOrder: 200+ KIPS
Simulation Suite Overview

- **Sim-Fast**
  - 420 lines
  - functional
  - 4+ MIPS

- **Sim-Safe**
  - 350 lines
  - functional
  - w/ checks

- **Sim-Profile**
  - 900 lines
  - functional
  - lot of stats

- **Sim-Cache/ Sim-Cheetah**
  - < 1000 lines
  - functional
  - cache stats

- **Sim-Outorder**
  - 3900 lines
  - performance
  - OoO issue
  - branch pred.
  - mis-spec.
  - ALUs
  - cache
  - TLB
  - 200+ KIPS

Performance

Detail
### Simulator Structure

- **Core Components:**
  - **SimpleScalar Program Binary**
  - **SimpleScalar ISA**
  - **POSIX System Calls**
  - **Machine Definition**
  - **Proxy Syscall Handler**
  - **BPred**
  - **Simulator Core**
  - **Stats**
  - **Dlite!**
  - **Resource**
  - **Cache**
  - **Loader**
  - **Regs**
  - **Memory**

- **Functionality:**
  - Modular components facilitate “rolling your own”
  - Performance core is optional

---

Todd M. Austin
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
Installation Notes

• follow the installation directions in the tech report, and *DON’T PANIC!!!!*

• avoid building GLIBC
  ◯ it’s a non-trivial process
  ◯ use the big- and little-endian, pre-compiled libraries in ss-bootstrap/

• if you have problems, send e-mail to the SimpleScalar mailing list: simplescalar@cs.wisc.edu

• please e-mail install mods to: dburger@cs.wisc.edu

• x86 port has limited functionality, portability
  ◯ currently not supported
  ◯ reportedly only works under little-endian Linux
Generating SimpleScalar Binaries

- compiling a C program, e.g.,
  \texttt{ssbig-na-sstrix-gcc -g -O -o foo foo.c -lm}
- compiling a Fortran program, e.g.,
  \texttt{ssbig-na-sstrix-f77 -g -O -o foo foo.f -lm}
- compiling a SimpleScalar assembly program, e.g.,
  \texttt{ssbig-na-sstrix-gcc -g -O -o foo foo.s -lm}
- running a program, e.g.,
  \texttt{sim-safe [-sim opts] program [-program opts]}
- disassembling a program, e.g.,
  \texttt{ssbig-na-sstrix-objdump -x -d -l foo}
- building a library, use:
  \texttt{ssbig-na-sstrix-\{ar,ranlib\}}
Global Simulator Options

- supported on all simulators:
  -h - print simulator help message
  -d - enable debug message
  -i - start up in DLite! debugger
  -q - terminate immediately (use with \texttt{-dumpconfig})
  -config \texttt{<file>} - read configuration parameters from \texttt{<file>}
  -dumpconfig \texttt{<file>} - save configuration parameters into \texttt{<file>}

- configuration files:
  - to generate a configuration file:
    - specify non-default options on command line
    - and, include \texttt{\texttt{-dumpconfig \texttt{<file>}}} to generate configuration file
  - comments allowed in configuration files:
    - text after “#” ignored until end of line
  - reload configuration files using \texttt{\texttt{-config \texttt{<file>}}}”
  - config files may reference other configuration files
DLite!, the Lite Debugger

- a very lightweight symbolic debugger
- supported by all simulators (except sim-fast)
- designed for easily integration into SimpleScalar simulators
  - requires addition of only four function calls (see dlite.h)
- to use DLite!, start simulator with “-i” option (interactive)
- program symbols and expressions may be used in most contexts
  - e.g., “break main+8”
- use the “help” command for complete documentation
- main features:
  - `break`, `dbreak`, `rbreak`: set text, data, and range breakpoints
  - `regs`, `iregs`, `fregs`: display all, int, and FP register state
  - `dump <addr> <count>`: dump `<count>` bytes of memory at `<addr>`
  - `dis <addr> <count>`: disassemble `<count>` insts starting at `<addr>`
  - `print <expr>, display <expr>`: display expression or memory
  - `mstate`: display machine-specific state
DLite!, the Lite Debugger (cont.)

- breakpoints:
  - code:
    - break <addr>
    - e.g., break main, break 0x400148
  - data:
    - dbreak <addr> {r|w|x}
    - r == read, w == write, x == execute
    - e.g., dbreak stdin w, dbreak sys_count wr
  - code:
    - rbreak <range>
    - e.g., rbreak @main:+279, rbreak 2000:3500

- DLite! expressions
  - operators: +, -, /, *
  - literals: 10, 0xff, 077
  - symbols: main, vfprintf
  - registers: $r1, $f4, $pc, $fcc, $hi, $lo
Execution Ranges

- specify a range of addresses, instructions, or cycles
- used by range breakpoints and pipetracer (in sim-outorder)
- format:

  address range:    @<start>:<end>
  instruction range: <start>:<end>
  cycle range:      #<start>:<end>

- the end range may be specified relative to the start range
- both endpoints are optional, and if omitted the value will default to the largest/smallest allowed value in that range
- e.g.,
  - @main:+278 - main to main+278
  - #:1000 - cycle 0 to cycle 1000
  - : - entire execution (instruction 0 to end)
Sim-Safe: Functional Simulator

- the minimal SimpleScalar simulator
- no other options supported
Sim-Fast: Fast Functional Simulator

- an optimized version of sim-safe
- DLite! is not supported on this simulator
- no other options supported
Sim-Profile: Program Profiling Simulator

- generates program profiles, by symbol and by address
- extra options:
  - `iclass` - instruction class profiling (e.g., ALU, branch)
  - `iprof` - instruction profiling (e.g., bnez, addi, etc...)
  - `brprof` - branch class profiling (e.g., direct, calls, cond)
  - `amprof` - address mode profiling (e.g., displaced, R+R)
  - `segprof` - load/store segment profiling (e.g., data, heap)
  - `tsymprof` - execution profile by text symbol (i.e., funcs)
  - `dsymprof` - reference profile by data segment symbol
  - `taddrprof` - execution profile by text address
  - `all` - enable all of the above options
  - `pcstat <stat>` - record statistic `<stat>` by text address

**NOTE:** “-taddrprof” == “-pcstat sim_num_insn”
PC-Based Statistical Profiles (-pcstat)

- produces a text segment profile for any integer statistical counter
- supported on sim-cache, sim-profile, and sim-outorder
- specify a statistical counter to be monitored using “-pcstat” option
  - e.g., -pcstat sim_num_insn
- example applications:
  -pcstat sim_num_insn - execution profile
  -pcstat sim_num.refs - reference profile
  -pcstat ill.misses - L1 I-cache miss profile (sim-cache)
  -pcstat bpred_bimod.misses - br pred miss profile (sim-outorder)

- view with the textprof.pl Perl script, it displays pc-based statistics with program disassembly:

  textprof.pl <dis_file> <sim_output> <stat_name>
PC-Based Statistical Profiles (cont.)

- **example usage:**

  ```
  sim-profile -pcstat sim_num_insn test-math >&! test-math.out
  objdump -dl test-math >&! test-math.dis
  textprof.pl test-math.dis test-math.out sim_num_insn_by_pc
  ```

- **example output:**

  ```
  00401a10:  ( 13, 0.01): <strtod+220> addiu $a1[5],$zero[0],1
              strconv.c:79
  00401a18:  ( 13, 0.01): <strtod+228> bc1f 00401a30 <strtod+240>
              strconv.c:87
  00401a20:               : <strtod+230> addiu $s1[17],$s1[17],1
  00401a28:               : <strtod+238> j 00401a58 <strtod+268>
              strconv.c:89
  00401a30:  ( 13, 0.01): <strtod+240> mul.d $f2,$f20,$f4
  00401a38:  ( 13, 0.01): <strtod+248> addiu $v0[2],$v1[3],-48
  00401a40:  ( 13, 0.01): <strtod+250> mtc1 $v0[2],$f0
  ```

- **works on any integer counter registered with the stats package, including those added by users!**
Sim-Cache: Multi-level Cache Simulator

- generates one- and two-level cache hierarchy statistics and profiles
- extra options (also supported on sim-outorder):
  - cache:dl1 <config> - level 1 data cache configuration
  - cache:dl2 <config> - level 2 data cache configuration
  - cache:il1 <config> - level 1 instruction cache configuration
  - cache:il2 <config> - level 2 instruction cache configuration
  - tlb:dtlb <config> - data TLB configuration
  - tlb:itlb <config> - instruction TLB configuration
  - flush <config> - flush caches on system calls
  - icompress - remaps 64-bit inst addresses to 32-bit equiv.
  - pcstat <stat> - record statistic <stat> by text address
Specifying Cache Configurations

• all caches and TLB configurations specified with same format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>

• where:

    <name> - cache name (make this unique)
    <nsets> - number of sets
    <assoc> - associativity (number of “ways”)
    <repl> - set replacement policy
        l - for LRU
        f - for FIFO
        r - for RANDOM

• examples:

    ill:1024:32:2:l  2-way set-assoc 64k-byte cache, LRU
    dtlb:1:4096:64:r  64-entry fully assoc TLB w/ 4k pages, random replacement
Specifying Cache Hierarchies

- specify all cache parameters in no unified levels exist, e.g.,

```
```

- to unify any level of the hierarchy, “point” an I-cache level into the data cache hierarchy:

```
-il1:128:64:1:1 -il2:dl2
```

Todd M. Austin
Sim-Cheetah: Multi-Config Cache Simulator

- generates cache statistics and profiles for multiple cache configurations in a single program execution
- uses Cheetah cache simulation engine
  - written by Rabin Sugumar and Santosh Abraham while at Umich
  - modified to be a standalone library, see “libcheetah/” directory
- extra options:
  - `refs {inst,data,unified} - specify reference stream to analyze`
  - `C {fa,sa,dm} - cache config. i.e., fully or set-assoc or direct`
  - `R {lru, opt} - replacement policy`
  - `a <sets> - log base 2 number of set in minimum config`
  - `b <sets> - log base 2 number of set in maximum config`
  - `l <line> - cache line size in bytes`
  - `n <assoc> - maximum associativity to analyze (log base 2)`
  - `in <interval> - cache size interval for fully-assoc analyses`
  - `M <size> - maximum cache size of interest`
  - `c <size> - cache size for direct-mapped analyses`
Sim-Outorder: Detailed Performance Simulator

- generates timing statistics for a detailed out-of-order issue processor core with two-level cache memory hierarchy and main memory
- extra options:

  - `fetch:ifqsize <size>` - instruction fetch queue size (in insts)
  - `fetch:mplat <cycles>` - extra branch mis-prediction latency (cycles)
  - `bpred <type>` - specify the branch predictor
  - `decode:width <insts>` - decoder bandwidth (insts/cycle)
  - `issue:width <insts>` - RUU issue bandwidth (insts/cycle)
  - `issue:inorder` - constrain instruction issue to program order
  - `issue:wrongpath` - permit instruction issue after mis-speculation
  - `ruu:size <insts>` - capacity of RUU (insts)
  - `lsq:size <insts>` - capacity of load/store queue (insts)
  - `cache:dll <config>` - level 1 data cache configuration
  - `cache:dll1lat <cycles>` - level 1 data cache hit latency
Sim-Outorder: Detailed Performance Simulator

- `cache:dl2 <config>` - level 2 data cache configuration
- `cache:dl2lat <cycles>` - level 2 data cache hit latency
- `cache:il1 <config>` - level 1 instruction cache configuration
- `cache:il1lat <cycles>` - level 1 instruction cache hit latency
- `cache:il2 <config>` - level 2 instruction cache configuration
- `cache:il2lat <cycles>` - level 2 instruction cache hit latency
- `cache:flush` - flush all caches on system calls
- `cache:icompress` - remap 64-bit inst addresses to 32-bit equiv.
- `mem:lat <1st> <next>` - specify memory access latency (first, rest)
- `mem:width` - specify width of memory bus (in bytes)
- `tlb:itlb <config>` - instruction TLB configuration
- `tlb:dtlb <config>` - data TLB configuration
- `tlb:lat <cycles>` - latency (in cycles) to service a TLB miss
Sim-Outorder: Detailed Performance Simulator

- `res:ialu` - specify number of integer ALUs
- `res:imult` - specify number of integer multiplier/dividers
- `res:memports` - specify number of first-level cache ports
- `res:fpalu` - specify number of FP ALUs
- `res:fpmult` - specify number of FP multiplier/dividers
- `pcstat <stat>` - record statistic `<stat>` by text address
- `ptrace <file> <range>` - generate pipetrace
Specifying the Branch Predictor

• specifying the branch predictor type:

   -bpred <type>

   the supported predictor types are:
   - nottaken always predict not taken
   - taken always predict taken
   - perfect perfect predictor
   - bimod bimodal predictor (BTB w/ 2 bit counters)
   - 2lev 2-level adaptive predictor

• configuring the bimodal predictor (only useful when “-bpred bimod” is specified):

   -bpred:bimod <size> size of direct-mapped BTB
Specifying the Branch Predictor (cont.)

• configuring the 2-level adaptive predictor (only useful when “-bpred 2lev” is specified):

  -bpred:2lev <l1size> <l2size> <hist_size>

where:

- `<l1size>` size of the first level table
- `<l2size>` size of the second level table
- `<hist_size>` history (pattern) width
Sim-Outorder Pipetraces

- produces detailed history of all instructions executed, including:
  - instruction fetch, retirement, and stage transitions
- supported in sim-outorder
- use the “-ptrace” option to generate a pipetrace
  - -ptrace <file> <range>
- example usage:

  -pcstat FOO.trc          - trace entire execution to FOO.trc
  -pcstat BAR.trc 100:5000 - trace from inst 100 to 5000
  -pcstat UXXE.trc :10000  - trace until instruction 10000

- view with the pipeview.pl Perl script, it displays the pipeline for each cycle of execution traced:

  pipeview.pl <ptrace_file>
Sim-Outorder Pipetraces (cont.)

- example usage:

```
sim-outorder -ptrace FOO.trc :1000 test-math
pipeview.pl FOO.trc
```

- example output:

```
@ 610
```

```
gf = '0x0040d098: addiu r2,r4,-1'
gg = '0x0040d0a0: beq r3,r5,0x30'
```

new cycle indicator

```
new inst definitions
```

```
[IF]
gf
[DA]
gb
[EX]
fy
[WB]
fr
[CT]
fq
```

current pipeline state

```
inst being fetched, or in fetch queue
```

```
inst being decoded, or awaiting issue
```

```
inst executing
```

```
inst writing results into RUU, or awaiting retire
```

pipeline event: (mis-prediction detected), see output header for event defs
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
The SimpleScalar Instruction Set

- clean and simple instruction set architecture:
  - MIPS/DLX + more addressing modes - delay slots
- bi-endian instruction set definition
  - facilitates portability, build to match host endian
- 64-bit inst encoding facilitates instruction set research
  - 16-bit space for hints, new insts, and annotations
  - four operand instruction format, up to 256 registers

```
+----------------+----------------+----------------+----------------+----------------+
| 16-annotate    | 16-opcode      | 8-ru           | 8-rt           | 16-imm         |
+----------------+----------------+----------------+----------------+----------------+
| 63             | 48             | 32             | 24             | 16             |
+----------------+----------------+----------------+----------------+----------------+----------------+
|                 |                |                |                | 8              |
+----------------+----------------+----------------+----------------+----------------+----------------+
|                 |                |                |                | 0              |
+----------------+----------------+----------------+----------------+----------------+----------------+```
SimpleScalar Architected State

Integer Reg File
- r0 - 0 source/sink
- r1 (32 bits)
- r2
- .
- .
- r30
- r31

FP Reg File (SP and DP views)
- f0 (32 bits)
- f1
- f2
- .
- .
- f30
- f31

Virtual Memory
- 0x00000000
- 0x7fffffff
- Unused
- Text (code)
- Data (init) (bss)
- Stack
- Args & Env

r0 - 0 source/sink
r1 (32 bits)
r2
.
.
r30
r31

f0 (32 bits)
f1
f2
.
.
f30
f31

Todd M. Austin
SimpleScalar Instructions

Control:
- j - jump
- jal - jump and link
- jr - jump register
- jalr - jump and link register
- beq - branch == 0
- bne - branch != 0
- blez - branch <= 0
- bgtz - branch > 0
- bltz - branch < 0
- bgez - branch >= 0
- bct - branch FCC TRUE
- bcf - branch FCC FALSE

Load/Store:
- lb - load byte
- lbu - load byte unsigned
- lh - load half (short)
- lhu - load half (short) unsigned
- lw - load word
- dlw - load double word
- l.s - load single-precision FP
- l.d - load double-precision FP
- sb - store byte
- sbu - store byte unsigned
- sh - store half (short)
- shu - store half (short) unsigned
- sw - store word
- dsw - store double word
- s.s - store single-precision FP
- s.d - store double-precision FP

Integer Arithmetic:
- add - integer add
- addu - integer add unsigned
- sub - integer subtract
- subu - integer subtract unsigned
- mult - integer multiply
- multu - integer multiply unsigned
- div - integer divide
- divu - integer divide unsigned
- and - logical AND
- or - logical OR
- xor - logical XOR
- nor - logical NOR
- sll - shift left logical
- srl - shift right logical
- sra - shift right arithmetic
- slt - set less than
- sltu - set less than unsigned

addressing modes:
(C)
(reg + C) (w/ pre/post inc/dec)
(reg + reg) (w/ pre/post inc/dec)
SimpleScalar Instructions

Floating Point Arithmetic:

- add.s - single-precision add
- add.d - double-precision add
- sub.s - single-precision subtract
- sub.d - double-precision subtract
- mult.s - single-precision multiply
- mult.d - double-precision multiply
- div.s - single-precision divide
- div.d - double-precision divide
- abs.s - single-precision absolute value
- abs.d - double-precision absolute value
- neg.s - single-precision negation
- neg.d - double-precision negation
- sqrt.s - single-precision square root
- sqrt.d - double-precision square root
- cvt - integer, single, double conversion
- c.s - single-precision compare
- c.d - double-precision compare

Miscellaneous:

- nop - no operation
- syscall - system call
- break - declare program error
Annotating SimpleScalar Instructions

- useful for adding
  - hints, new instructions, text markers, etc...
  - no need to hack the assembler
- bit annotations:
  - /a - /p, set bit 0 - 15
  - e.g., 1d/a $r6,4($r7)
- field annotations:
  - /s:e(v), set bits s->e with value v
  - e.g., 1d/6:4(7) $r6,4($r7)
Proxy System Call Handler

- syscall.c implements a subset of Ultrix Unix system calls
- basic algorithm:
  - decode system call
  - copy arguments (if any) into simulator memory
  - make system call
  - copy results (if any) into simulated program memory
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
Simulator Structure

- modular components facilitate “rolling your own”
- performance core is optional
Out-of-Order Issue Simulator

- implemented in `sim-outorder.c` and modules
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead
Out-of-Order Issue Simulator: Fetch

- implemented in `ruu_fetch()`
- models machine fetch bandwidth
- inputs:
  - program counter
  - predictor state (see `bpred.[hc]`)
  - mis-prediction detection from branch execution unit(s)
- outputs:
  - fetched instructions to Dispatch queue
Out-of-Order Issue Simulator: Fetch

- procedure (once per cycle):
  - fetch insts from *one* I-cache line, block until misses are resolved
  - queue fetched instructions to Dispatch
  - probe line predictor for cache line to access in next cycle
Out-of-Order Issue Simulator: Dispatch

- implemented in `ruu_dispatch()`
- models machine decode, rename, allocate bandwidth
- inputs:
  - instructions from input queue, fed by Fetch stage
  - RUU
  - rename table (`create_vector`)
  - architected machine state (for execution)
- outputs:
  - updated RUU, rename table, machine state
Out-of-Order Issue Simulator: Dispatch

- procedure (once per cycle):
  - fetch insts from Dispatch queue
  - decode and *execute* instructions
    - facilitates simulation of data-dependent optimizations
    - permits early detection of branch mis-predicts
  - if mis-predict occurs:
    - start copy-on-write of architected state to speculative state buffers
  - enter and link instructions into RUU and LSQ (load/store queue)
    - links implemented with RS_LINK structure
    - loads/stores are split into two insts: ADD → Load/Store
    - improves performance of memory dependence checking
Out-of-Order Issue Simulator: Scheduler

- implemented in ruu_issue() and lsq_refresh()
- models instruction, wakeup, and issue to functional units
  - separate schedulers to track register and memory dependencies
- inputs:
  - RUU, LSQ
- outputs:
  - updated RUU, LSQ
  - updated functional unit state
Out-of-Order Issue Simulator: Scheduler

- procedure (once per cycle):
  - locate instructions with all register inputs ready
    - in ready queue, inserted during dependent inst’s wakeup walk
  - locate instructions with all memory inputs ready
    - determined by walking the load/store queue
    - if earlier store with unknown addr → stall issue (and poll)
    - if earlier store with matching addr → store forward
    - else → access D-cache
Out-of-Order Issue Simulator: Execute

- implemented in `ruu_issue()`
- models func unit and D-cache issue and execute latencies
- inputs:
  - ready insts as specified by Scheduler
  - functional unit and D-cache state
- outputs:
  - updated functional unit and D-cache state
  - updated event queue, events notify Writeback of inst completion
Out-of-Order Issue Simulator: Execute

issued insts from Scheduler → Exec → finished insts to Writeback

Mem

memory requests to D-cache

• procedure (once per cycle):
  • get ready instructions (as many as supported by issue B/W)
  • probe functional unit state for availability and access port
  • reserve unit it can issue again
  • schedule writeback event using operation latency of functional unit
    • for loads satisfied in D-cache, probe D-cache for access latency
    • also probe D-TLB, stall future issue on a miss
    • D-TLB misses serviced at commit time with fixed latency
Out-of-Order Issue Simulator: Writeback

detected mis-prediction to Fetch

finished insts from Execute → Writeback → insts ready to commit to Commit

• implemented in ruu_writeback()
• models writeback bandwidth, detects mis-predictions, initiated mis-prediction recovery sequence
• inputs:
  - completed instructions as indicated by event queue
  - RUU, LSQ state (for wakeup walks)
• outputs:
  - updated event queue
  - updated RUU, LSQ, ready queue
  - branch mis-prediction recovery updates
Out-of-Order Issue Simulator: Writeback

detected mis-prediction to Fetch

finished insts from Execute \rightarrow Writeback \rightarrow insts ready to commit to Commit

- procedure (once per cycle):
  - get finished instructions (specified in event queue)
  - if mis-predicted branch:
    - recover RUU
      - walk newest inst to mis-pred branch
      - unlink insts from output dependence chains
    - recover architected state
      - roll back to checkpoint
  - wakeup walk: walk dependence chains of inst outputs
    - mark dependent inst’s input as now ready
    - if all reg dependencies of the dependent inst are satisfied, wake it up (memory dependence check occurs later in Issue)
Out-of-Order Issue Simulator: Commit

insts ready to commit from Writeback → Commit

- implemented in `ruu_commit()`
- models in-order retirement of instructions, store commits to the D-cache, and D-TLB miss handling
- inputs:
  - completed instructions in RUU/LSQ that are ready to retire
  - D-cache state (for committed stores)
- outputs:
  - updated RUU, LSQ
  - updated D-cache state
Out-of-Order Issue Simulator: Commit

insts ready to commit from Writeback → Commit

- procedure (once per cycle):
  - while head of RUU is ready to commit (in-order retirement)
    - if D-TLB miss, then service it
    - then if store, attempt to retire store into D-cache, stall commit otherwise
    - commit inst result to the architected register file, update rename table to point to architected register file
    - reclaim RUU/LSQ resources
Out-of-Order Issue Simulator: Main

```c
ruu_init()
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsqRefresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

- implemented in `sim_main()`
- walks pipeline from Commit to Fetch
  - backward pipeline traversal eliminates relaxation problems, e.g.,
    provides correct inter-stage latch synchronization
- loop is execute via a `longjmp()` to `main()` when simulated
  program executes an `exit()` system call
Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User’s Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- **Hacking SimpleScalar**
- Looking Ahead
Hacker’s Guide

• source code design philosophy:
  • infrastructure facilitates “rolling your own”
    • standard simulator interfaces
    • large component library, e.g., caches, loaders, etc...
  • performance and flexibility before clarity

• section organization:
  • compiler chain hacking
  • simulator hacking
Hacking the Compiler (GCC)

- see GCC.info in the GNU GCC release for details on the internals of GCC
- all SimpleScalar-specific code is in the config/ss in the GNU GCC source tree
- use instruction annotations to add new instruction, as you won’t have to then hack the assembler
- avoid adding new linkage types, or you will have to hack GAS, GLD, and libBFD.a, all of which are very painful
Hacking the Assembler (GAS)

• most of the time, you should be able to avoid this by using instruction annotations
• new instructions are added in libopcode.a, new instructions will also be picked up by disassembler
• new linkage types require hacking GLD and libBFD.a, which is very painful
Hacking the Linker (GLD and libBFD.a)

- avoid this if possible, both tools are difficult to comprehend and generally delicate
- if you must...
  - emit a linkage map (-Map mapfile) and then edit the executable in a postpass
  - KLINK, from my dissertation work, does exactly this
Hacking the SimpleScalar Simulators

- two options:
  - leverage existing simulators (sim-*.c)
    - they are stable
    - very little instrumentation has been added to keep the source clean
  - roll your own
    - leverage the existing simulation infrastructure, i.e., all the files that do not start with ‘sim-’
    - consider contributing useful tools to the source base
- for documentation, read interface documentation in “.h” files
Simulator Structure

- modular components facilitate “rolling your own”
- performance core is optional
Machine Definition

- A single file describes all aspects of the architecture
  - Used to generate decoders, dependency analyzers, functional components, disassemblers, appendices, etc.
  - E.g., machine definition + 10 line main == functional sim
  - Generates fast and reliable codes with minimum effort

- Instruction definition example:

```
DEFINST(ADDI, 0x41, "addi", "t,s,i", IntALU, F_ICOMP|F_IMM, GPR(RT), NA, GPR(RS), NA, NA, SET_GPR(RT, GPR(RS)+IMM))
```
Crafting a Functional Component

```c
#define GPR(N)                  (regs_R[N])
#define SET_GPR(N,EXPR)         (regs_R[N] = (EXPR))
#define READ_WORD(SRC, DST)     (mem_read_word((SRC))

switch (SS_OPCODE(inst)) {
    #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3,EXPR)
        case OP:
            EXPR;                                                
        break;
    #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
        case OP:
            panic("attempted to execute a linking opcode");
    #define CONNECT(OP)

    #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3,EXPR) \ 
    case OP: \ 
        EXPR; \ 
        break; \ 
    #define DEFLINK(OP,MSK,NAME,MASK,SHIFT) \ 
        case OP: \ 
            panic("attempted to execute a linking opcode"); \ 
    #define CONNECT(OP) \ 
    #include "ss.def" \ 
    #undef DEFINST \ 
    #undef DEFINST \ 
    #undef DEFLINK \ 
    #undef CONNECT \ 
}
```
Crafting an Decoder

```c
#define DEP_GPR(N)              (N)

switch (SS_OPCODE(inst)) {
    #define DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,O1,O2,I1,I2,I3,EXPR)   
      case OP:                                                      
        out1 = DEP_##O1; out2 = DEP_##O2;                           
        in1 = DEP_##I1; in2 = DEP_##I2; in3 = DEP_##I3;             
        break;
    #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                             
      case OP:                                 ...                                      
        in1 = NA; in2 = NA; in3 = NA;                               
        break;
    #define CONNECT(OP)
    #include "ss.def"
    #undef DEFINST
    #undef DEFLINK
    #undef CONNECT
    default:        /* can speculatively decode a bogus inst */        
        op = NOP;        out1 = NA; out2 = NA;
        in1 = NA; in2 = NA; in3 = NA;
        break;
    }
```
Options Module (option.[hc])

- options are registers (by type) into an options data base
  - see opt_reg_*() interfaces
- produce a help listing:
  - opt_print_help()
- print current options state:
  - opt_print_options()
- add a header to the help screen:
  - opt_reg_header()
- add notes to an option (printed on help screen):
  - opt_reg_note()
Stats Package (stats.[hc])

- one-stop shopping for statistical counters, expressions, and distributions
- counters are “registered” by type with the stats package:
  - see stat_reg_*() interfaces
  - stat_reg_formula(): register a stat that is an expression of other stats
    - stat_reg_formula(sdb, “ipc”, “insts per cycle”, “insns/cycles”, 0);
- simulator manipulates counters using standard in code, e.g.,
  - stat_num_insn++;
- stat package prints all statistics (using canonical format)
  - stat_print_stats()
- distributions also supported:
  - stat_reg_dist(): register an array distribution
  - stat_reg_sdist(): register a sparse distribution
  - stat_add_sample(): add a sample to a distribution
Proxy Syscall Handler (syscall.[hc])

• algorithm:
  - decode system call
  - copy arguments (if any) into simulator memory
  - make system call
  - copy results (if any) into simulated program memory

• you’ll need to hack this module to:
  - add new system call support
  - port SimpleScalar to an unsupported host OS
Branch Predictors (bpred.[hc])

- various branch predictors
  - static
  - BTB w/ 2-bit saturating counters
  - 2-level adaptive
- important interfaces:
  - bpred_create(class, size)
  - bpred_lookup(pred, br_addr)
  - bpred_update(pred, br_addr, targ_addr, result)
Cache Module (cache.[hc])

- ultra-vanilla cache module
  - can implement low- and high-assoc, caches, TLBs, etc...
  - efficient for all geometries
  - assumes a single-ported, fully pipelined backside bus
- important interfaces:
  - cache_create(name, nsets, bsize, balloc, usize, assoc repl, blk_fn, hit_latency)
  - cache_access(cache, op, addr, ptr, nbytes, when, udata)
  - cache_probe(cache, addr)
  - cache_flush(cache, when)
  - cache_flush_addr(cache, addr, when)
Event Queue (event.[hc])

- generic event (priority) queue
  - queue event for time t
  - returns events from the head of the queue
- important interfaces:
  - eventq_queue(when, op...)
  - eventq_service_events(when)
Program Loader (loader.[hc])

- prepares program memory for execution
  - loads program text
  - loads program data sections
  - initializes BSS section
  - sets up initial call stack
- important interfaces:
  - ld_load_prog(mem_fn, argc, argv, envp)
Main Routine (main.c, sim.h)

- defines interface to simulators
- important (imported) interfaces:
  - sim_options(argc, argv)
  - sim_config(stream)
  - sim_main()
  - sim_stats(stream)
Physical/Virtual Memory (memory.[hc])

- implements large flat memory spaces in simulator
  - uses single-level page table
  - may be used to implement virtual or physical memory
- important interfaces:
  - mem_access(cmd, addr, ptr, nbytes)
Miscellaneous Functions (misc.[hc])

- lots of useful stuff in here, e.g.:
  - fatal()
  - panic()
  - warn()
  - info()
  - debug()
  - getcore()
  - elapsed_time()
  - getopt()
Register State (regs.[hc])

- architected register variable definitions
Resource Manager (resource.[hc])

- powerful resource manager
  - configure with a resource pool
  - manager maintains resource availability
- resource configuration:
  
  ```
  { "name", num, { FU_class, issue_lat, op_lat }, ... }
  ```
- important interfaces:
  - res_create_pool(name, pool_def, ndefs)
  - res_get(pool, FU_class)
Tutorial Overview

• Computer Architecture Simulation Primer
• SimpleScalar Tool Set
  • Overview
  • User’s Guide
• SimpleScalar Instruction Set Architecture
• Out-of-Order Issue Simulator
  • Model Microarchitecture
  • Implementation Details
• Hacking SimpleScalar
• Looking Ahead
Looking Ahead

• MP/MT support for SimpleScalar simulators
• Linux port to SimpleScalar
  □ with device-level emulation and user-level file system
• Alpha and SPARC target support (SimpleScalar and MIPS currently exist)
To Get Plugged In

- SimpleScalar public releases available from UW-Madison
  - Public Release 2 is available from:
- Technical Report:
    UW-Madison Tech Report #1308, July 1996
- SimpleScalar mailing list:
  - simplescalar@cs.wisc.edu
  - contact Doug Burger (dburger@cs.wisc.edu) to join
Backups
Experiences and Insights

- the history of SimpleScalar:
  - Sohi’s CSim begat Franklin’s MSim begat SimpleScalar
  - first public release in July ‘96, made with Doug Burger
- key insights:
  - major investment req’d to develop sim infrastructure
    - 2.5 years to develop, while at UW-Madison
  - modular component design reduces design time and complexity, improves quality
  - fast simulators improve the design process, although it does introduce some complexity
  - virtual target improves portability, but limits workload
  - execution-driven simulation is worth the trouble
Advantages of Execution-Driven Simulation

- execution-based simulation
  - faster than tracing
    - fast simulators: 2+ MIPS, fast disks: < 1 MIPS
  - no need to store traces
  - register and memory values usually not in trace
    - functional component maintains precise state
    - extends design scope to include data-value-dependent optimizations
  - support mis-speculation cost modeling
    - on control and data dependencies
  - may be possible to eliminate most execution overheads
Example Applications

• my dissertation: “H/W and S/W Mechanisms for Reducing Load Latency”
  □ fast address calculation
  □ zero-cycle loads
  □ high-bandwidth address translation
  □ cache-conscious data placement

• other users:
  □ SCI project
  □ Galileo project
  □ more coming on-line
Related Tools

- SimOS from Stanford
  - includes OS and device simulation, and MP support
  - little source code since much of the tool chain is commercial code, e.g., compiler, operating system
  - not portable, currently only runs on MIPS hosts
- functional simulators:
  - direct execution via dynamic translation: Shade, FX32!
  - direct execution via static translation: Atom, EEL, Pixie
  - machine interpreters: Msim, DLXSim, Mint, AINT
Fast Functional Simulator

\texttt{sim\_main()}

\begin{center}
\begin{tikzpicture}
\node [draw, fill=gray!20] (0) at (0,0) {Opcode Map};
\node [draw, fill=cyan] (1) at (1,1) {ADDI};
\node [draw, fill=cyan] (2) at (1,0) {BNE};
\node [draw, fill=cyan] (3) at (1,-1) {FADD};
\node [draw, fill=cyan] (4) at (1,-2) {\ldots};
\draw [->] (0) -- (1);
\draw [->] (0) -- (2);
\draw [->] (0) -- (3);
\draw [->] (0) -- (4);
\end{tikzpicture}
\end{center}