

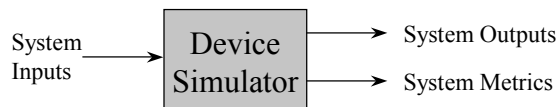
SimpleScalar Overview

Slides borrowed with permission from Todd Austin
info@simplescalar.com
SimpleScalar LLC

**SimpleScalar
LLC**

A Computer Architecture Simulator Primer

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

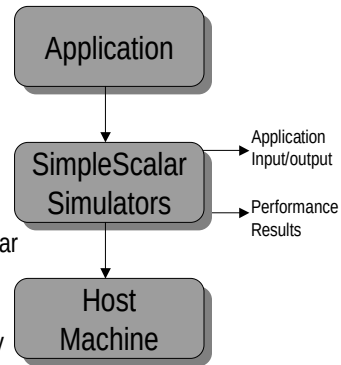


- Why use a simulator?
 - leverage faster, more flexible S/W development cycle
 - permits more design space exploration
 - facilitates validation before H/W becomes available
 - level of abstraction can be throttled to design task
 - possible to increase/improve system instrumentation

**SimpleScalar
LLC**

SimpleScalar Tool Set

- Computer system design and analysis infrastructure
 - Processor/device (behavioral) models
 - Supports many ISAs and I/O interfaces
 - Portable to most modern platforms
- Created by the SimpleScalar development team
 - UM, UW-Madison, UT-Austin, SimpleScalar LLC
 - Entering tenth year of development
 - Deployed widely in academia and industry
- Freely available with source and docs from www.simplescalar.com



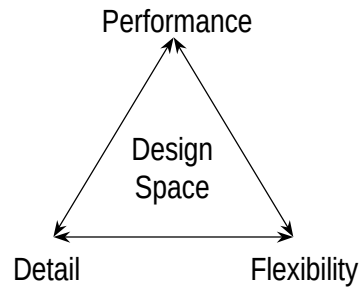
SimpleScalar
LLC

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 P4-1.7GHz)
 - Sim-Fast: 10+ MIPS
 - Sim-OutOrder: 350+ KIPS

SimpleScalar
LLC

The Zen of Hardware Model Design



Performance: speeds design cycle

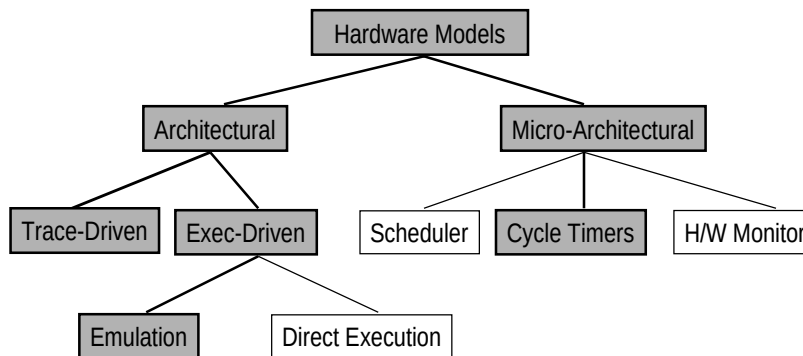
Flexibility: maximizes design scope

Detail: minimizes risk

- Infrastructure goals will drive which aspects are optimized
- SimpleScalar favors performance and flexibility

SimpleScalar
LLC

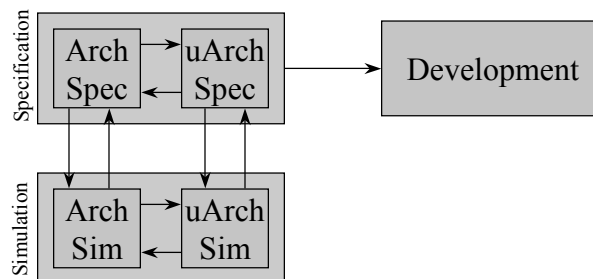
A Taxonomy of Hardware Modeling Tools



- Shaded tools are included in the SimpleScalar tool set

SimpleScalar
LLC

Functional vs. Performance Simulators

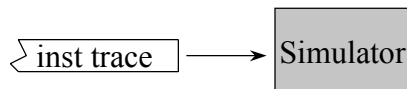


- functional simulators implement the architecture
 - the architecture is what programmer's see
- performance simulators implement the microarchitecture
 - model system internals (microarchitecture)
 - often concerned with time

SimpleScalar
LLC

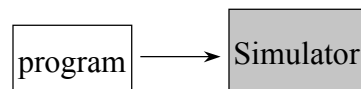
Execution- vs. Trace-Driven Simulation

- 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

SimpleScalar
LLC

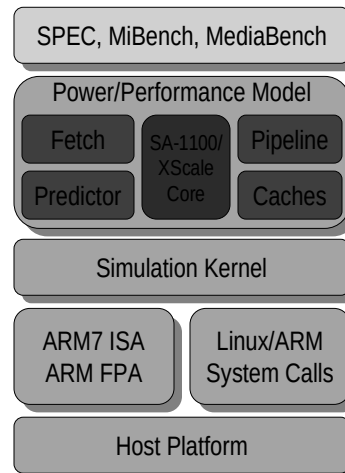
Cycle Level Simulator

- 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

SimpleScalar
LLC

SimpleScalar/ARM Target

- ARM simulation target
 - Developed by Dan Ernst and Chris Weaver
- ARM7 apps run on emulator
 - SPEC, MiBench, MediaBench
- Linux system call I/O emulator
 - Supports file, network, console I/O
- Multiple validated processor models
 - Intel StrongARM SA-1110
 - Intel XScale 80200
 - Performance and power models validated

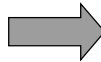


SimpleScalar
LLC

ARM Target Instruction Emulation

- ARM ISA emulation support added to SimpleScalar tool set
 - ARM 7 integer instruction set support
 - Floating Point Accelerator (FPA) instruction set support
- Linux/ARM system call support added
 - System calls are implemented by the simulator
 - Portable I/O, but does not capture OS execution
- ARM CISC instructions required microcode support
 - Needed for microarchitectural modeling

stmdb r13!,{r4-r8,r10-r15}

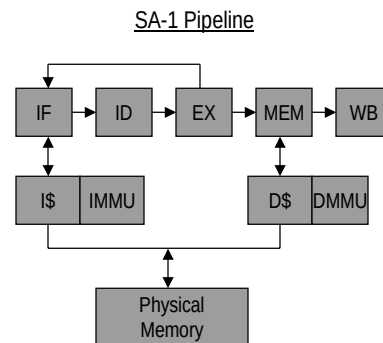


```
agen tmp1,r13,0
agen tmp0,tmp1,-16
stp r11,[tmp0]
agen r13,r13,-16
agen tmp0,tmp1,-12
stp r12,[tmp0]
agen tmp0,tmp1,-8
stp r14,[tmp0]
agen tmp0,tmp1,-4
stp r15,[tmp0]
```

SimpleScalar
LLC

Processor Performance Model

- SA-1 pipeline model implemented
 - Pipeline used in Intel's SA-11xx
 - Simple five stage pipeline
 - Two level memory hierarchy
- Challenging task due to lack of info on SA-1 microarchitecture
 - Derived many details from the compiler writers guide
 - Used directed black-box testing to fill in the rest of the blanks
- prototype XScale model completed
 - Intel's new StrongARM processor
 - Based on (sparse) published details
 - Validation ongoing against XScale 80200 evaluation board



SimpleScalar
LLC

ARM Cross-Compiler Kit

- Permits users to compile ARM binaries w/o ARM hardware
 - Most users lack access to a real ARM target with a native compiler
 - We use Rebel.com's NetWinder platforms to build native binaries
- GNU GCC targeted to ARM ISA
 - includes soft-float support (permits compilation for non-FP hardware)
- GNU binutils targeted to ARM ISA
 - GNU ld linker
 - GNU binary utilities, e.g., objdump, nm, size, etc...
- Pre-built C libraries for ARM ISA
 - Targeted to Linux system call interfaces
- Portable code base

SimpleScalar
LLC

Performance Model Validation

- Performance validation against SA-1110 platform
 - Rebel.com NetWinder reference with SA-1 pipeline
 - Microbenchmarks were used to reveal and test specific latencies
 - e.g., branch mispredictions, cache misses, writeback stalls
 - Final validation completed with macrobenchmark testing
 - Compared IPC of SA-1110 to IPCs computed by SA-1 performance model
 - H/W IPCs computed using wall clock time, clock frequency, and known instruction counts
 - Excellent IPC correlation across entire test suite

Benchmark	SimpleScalar	SA-1110	% Difference
cache_hit	1.02	1.01	0.9
cache_miss	33.87	33.70	0.5
br_taken	1.04	1.02	1.9
br_nottaken	1.97	1.91	3.1
bzip2 10	3.20	3.10	3.2
cc1 -O ccln.i	2.84	2.90	2.1
fft short.pcm	1.45	1.44	0.1

SimpleScalar
LLC

microbenchmarks
macrobenchmarks

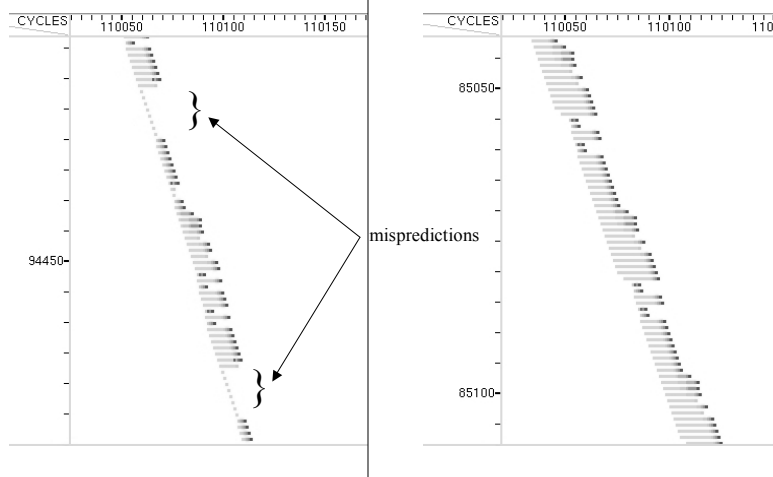
Sample Software Optimization: Loop Unrolling

- SA-110 ARM Model
 - Predict not taken
 - Multi-cycle mispredict per iteration
- 24% speed improvement using optimization

```
for (ii=38; ii >= 4; ii-=2)
{
  x = (D+D+1);
  w = (B+B+1);
  t = x*D;
  u = w*B;
  t = CONST_ROTLL(t, 5);
  u = CONST_ROTLL(u, 5);
  C -= S[ii];
  A -= S[ii+1];
  C = ROTR(C, u)^t;
  A = ROTR(A, t)^u;
  if (ii==4)
  { tmp = A; A = B; B = C; C = D; D = tmp;
  }
  else
  { tmp = A; A = D; D = C; C = B; B = tmp;
  }
}
```

SimpleScalar
LLC

Base vs. Optimized



SimpleScalar
LLC

MiBench Benchmark Suite

- Unencumbered embedded benchmark suite
 - Includes source code and multiple benchmark inputs
 - With binaries compiled for SimpleScalar/ARM simulator
 - Preliminary report details benchmarks and performance characteristics
- Six embedded programming domains (37 benchmarks)
 - Automotive/Industrial
 - Process control kernels from engine control, sensor monitoring
 - Networking/Security
 - Shortest path router, Patricia tree, packet processor, CRC32
 - Private and Public key ciphers, digest routines
 - 3DES, Blowfish, SHA, AES finalists
 - Consumer
 - Multimedia, image processing, entertainment
 - JPEG, Dither, RGBA, MediaBench, DOOM
 - Office
 - Spell, Grep, Ghostscript Postscript Interpreter
 - Telecommunications
 - FFT, GSM, ADPCM

SimpleScalar
LLC

Benchmark Categories

- Automotive & Industrial
 - Embedded control systems with sensor and actuator type applications.
- Consumer
 - Consumer devices like cameras, PDAs, scanners, etc.
- Office
 - Embedded office machinery like printers, organizers, word processors, etc.
- Network
 - Network devices such as switches, routers, and firewalls.
- Security
 - Encryption, decryption, hashing, and public key cryptography.
- Telecommunications
 - Algorithms for encoding and decoding communications.

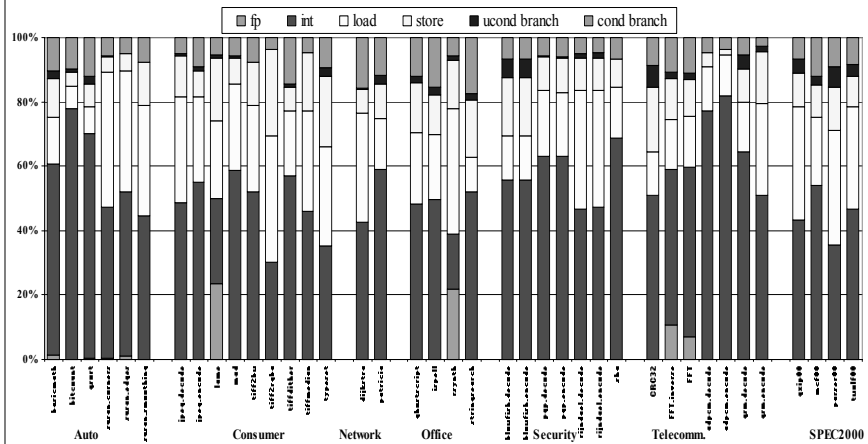
SimpleScalar
LLC

Benchmarks

Auto/Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg enc/dec	ghostscript	dijkstra	blowfish enc/dec	CRC32
bitcount	lame	ispell	patricia	pgp sign	FFT
qsort	mad	rsynth	(CRC32)	pgp verify	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	rijndael enc/dec	ADPCM enc/dec
susan (corners)	tiff2rgba	stringsearch	(blowfish)	sha	GSM enc/dec
susan (smoothing)	tiffdither				
	tiffmedian				
	typeset				

SimpleScalar
LLC

Instruction Distribution



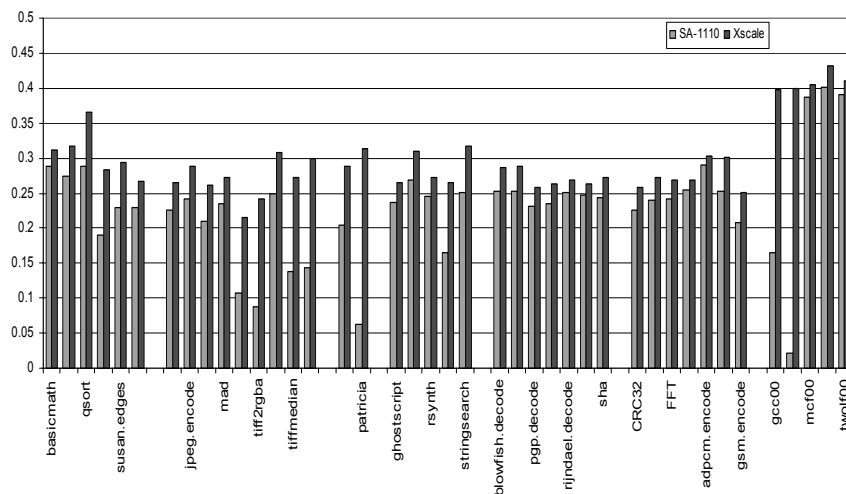
SimpleScalar
LLC

ARM Configurations

	SA-1100	Xscale
Fetch queue (instructions)	2	4
Branch Predictor	Not-taken	8k bimodal, 2k 4-way BTB
Fetch & Decode width	1	1
Functional Units	1 int ALU, 1 FP mult, 1 FP ALU	1 int ALU, 1 FP mult, 1 FP ALU
L1 I-cache	16k, 32-way	32k, 32-way
L1 D-cache	16k, 32-way	32k, 32-way
L2 Cachd	None	None
Memory Bus Width	4-byte	4-byte
Memory Latency	12 cycle	12 cycle

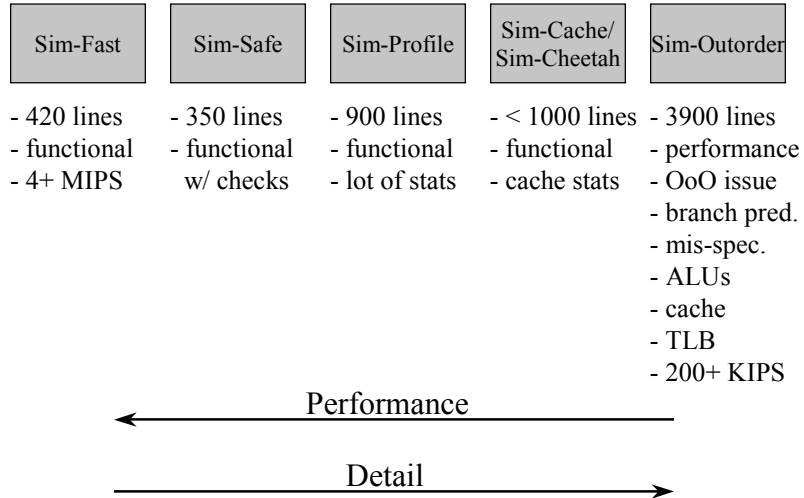
**SimpleScalar
LLC**

Achieved IPC



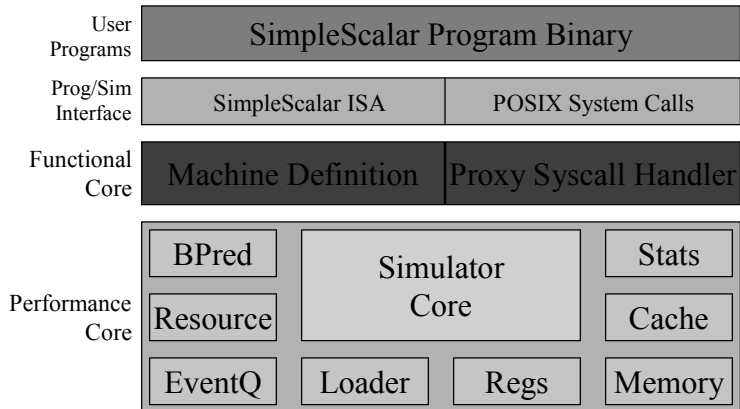
**SimpleScalar
LLC**

Simulation Suite Overview



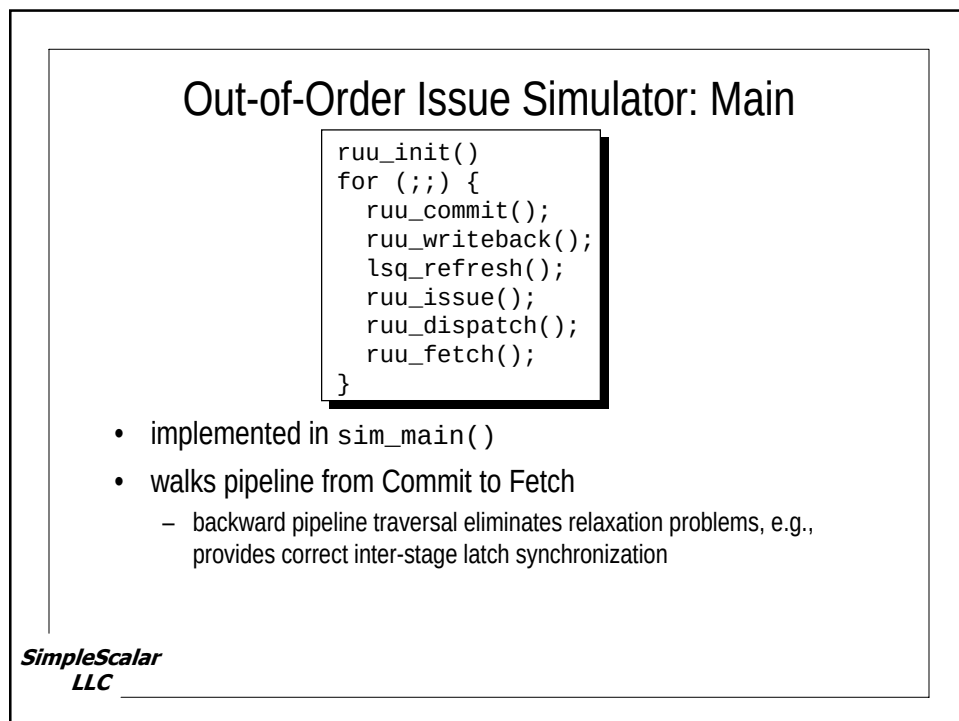
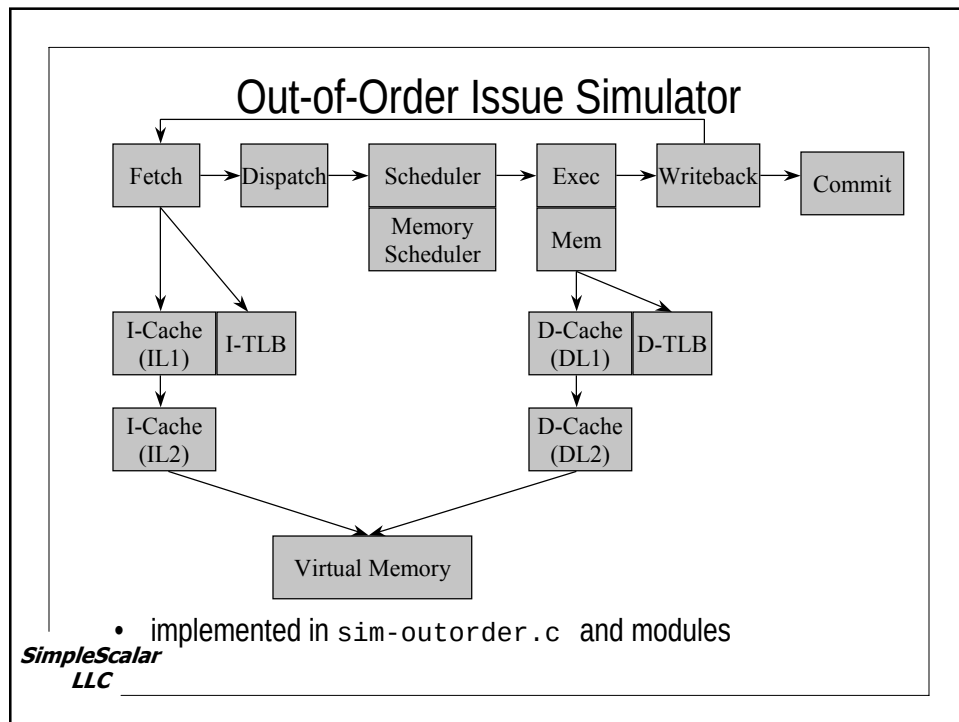
**SimpleScalar
LLC**

Simulator Structure

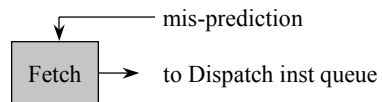


- modular components facilitate “rolling your own”
- performance core is optional

**SimpleScalar
LLC**



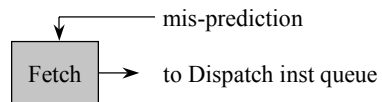
Out-of-Order Issue Simulator: Fetch



- 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

SimpleScalar
LLC

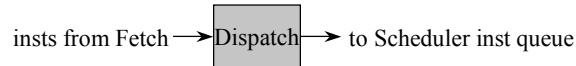
Out-of-Order Issue Simulator: Fetch



- 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

SimpleScalar
LLC

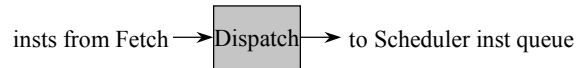
Out-of-Order Issue Simulator: Dispatch



- 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

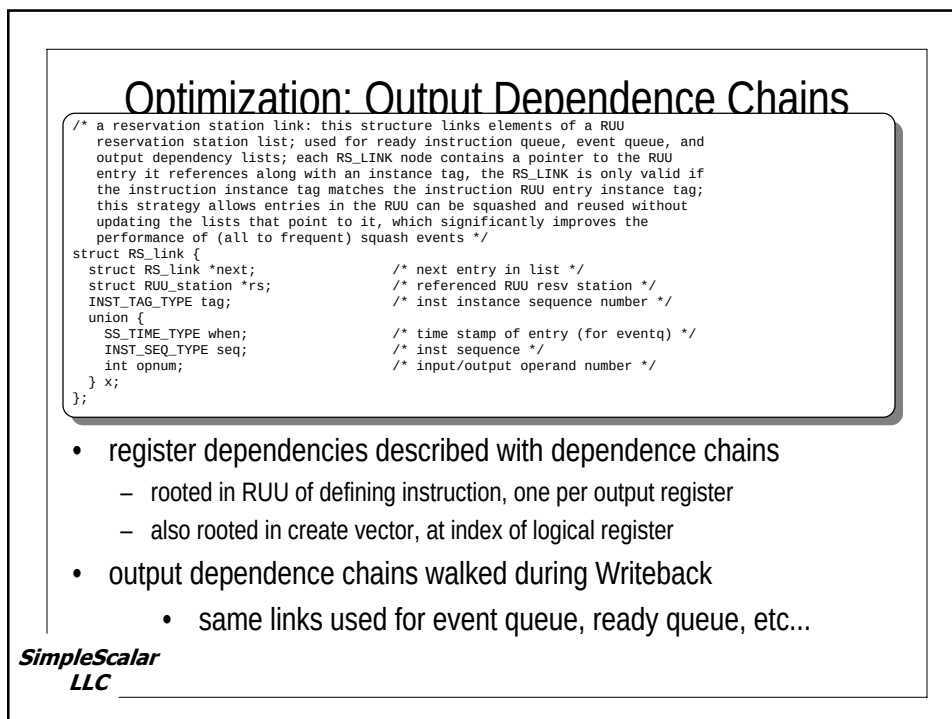
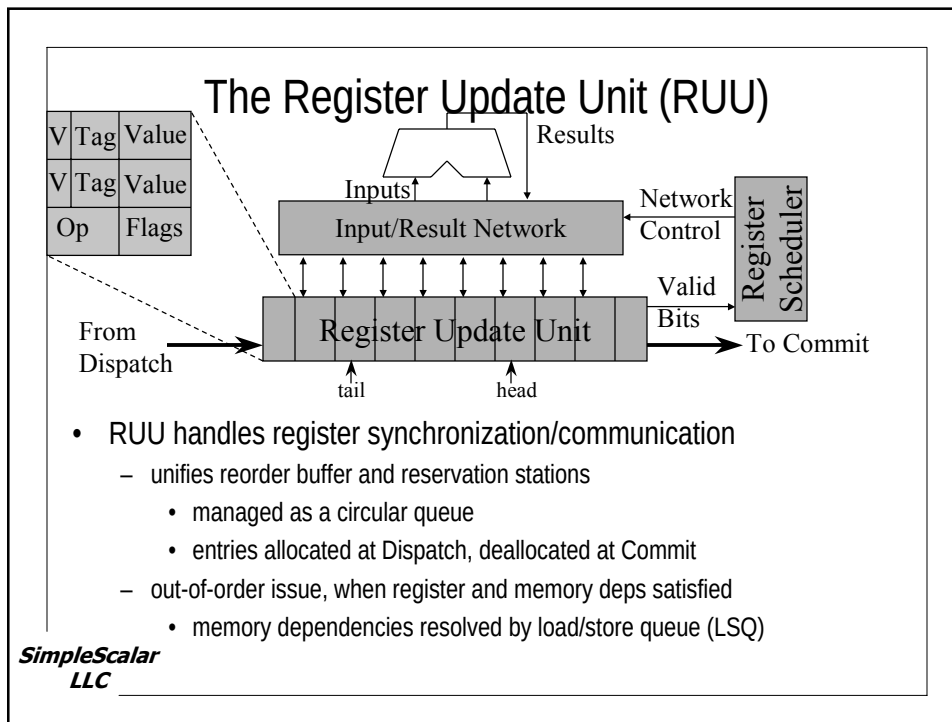
SimpleScalar
LLC

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`
 - speeds up memory dependence checking

SimpleScalar
LLC



Optimization: Output Dependence Chains

```

/* link RS onto the output chain number of whichever operation will create reg */
static INLINE void ruu_link_iddep(struct RUU_station *rs, int idep_num, int idep_name)
{
    struct CV_link head; struct RS_link *link;

    /* any dependence? */
    if (idep_name == NA) {
        /* no input dependence for this input slot, mark operand as ready */
        rs->idep_ready[idep_num] = TRUE;
        return;
    }

    /* locate creator of operand */
    head = CREATE_VECTOR(idep_name);

    /* any creator? */
    if (!head.rs) {
        /* no active creator, use value available in architected reg file,
         indicate the operand is ready for use */
        rs->idep_ready[idep_num] = TRUE;
        return;
    }
    /* else, creator operation will make this value sometime in the future */

    /* indicate value will be created sometime in the future, i.e., operand
     is not yet ready for use */
    rs->idep_ready[idep_num] = FALSE;

    /* link onto creator's output list of dependant operand */
    RSLINK_NEW(link, rs); link->x.opnum = idep_num;
    link->next = head.rs->odep_list[head.odep_num];
    head.rs->odep_list[head.odep_num] = link;
}

```

**SimpleScalar
LLC**

Optimization: Tagged Dependence Chains

- observation: “squash” recovery consumes many cycles
 - leverage “tagged” pointers to speed squash recover
 - unique tag assigned to each instruction, copied into references
 - squash an entry by destroying tag, makes all references stale

```

/* in ruu_recover(): squash this RUU entry */
RUU[RUU_index].tag++;

```

- all dereferences must check for stale references

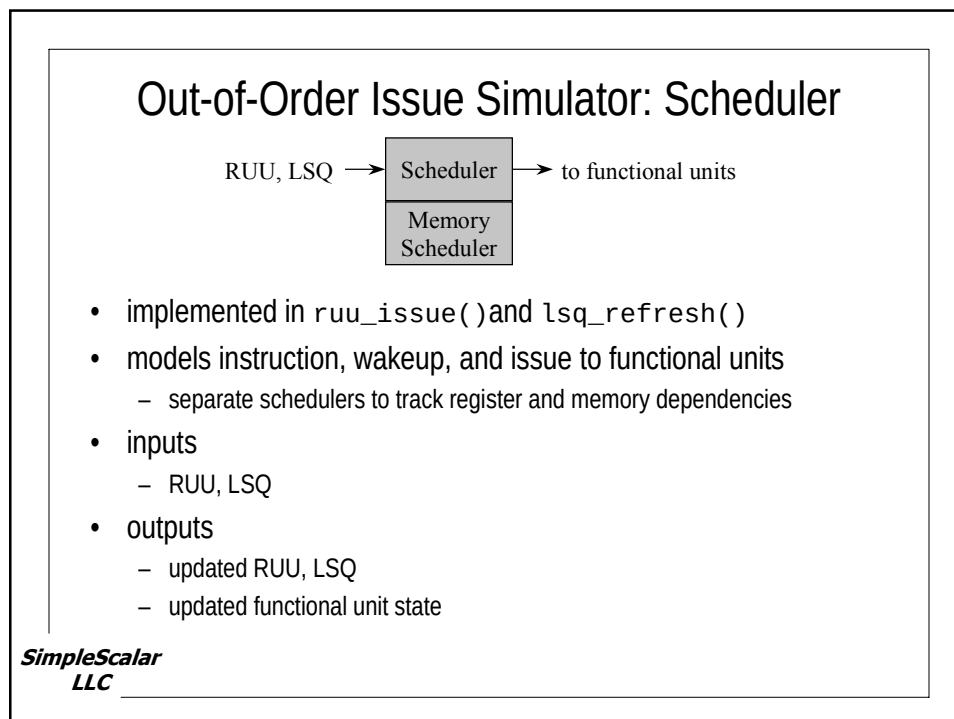
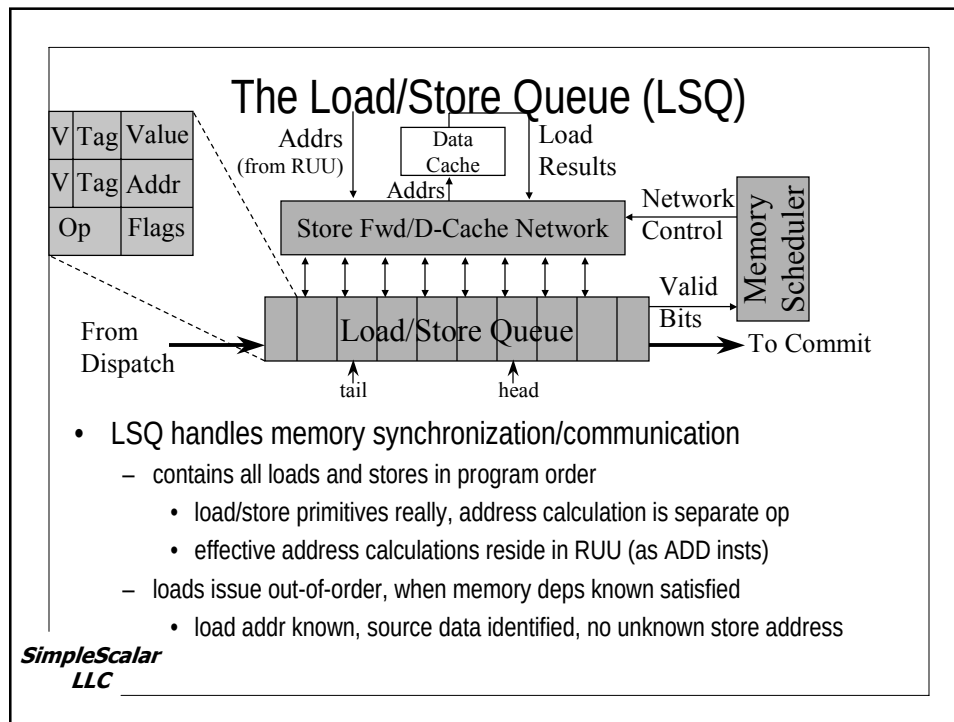
```

/* walk output list, queue up ready operations */
for (olink=rs->odep_list[i]; olink; olink=olink_next)
{
    if (RSLINK_VALID(olink)) {
        /* input is now ready */
        olink->r->idep_ready[olink->x.opnum] = TRUE;
    }
    . . .

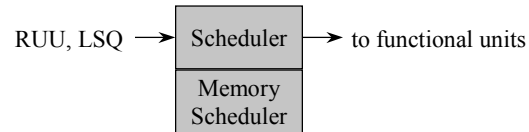
    /* grab link to next element prior to free */
    olink_next = olink->next;
}

```

**SimpleScalar
LLC**



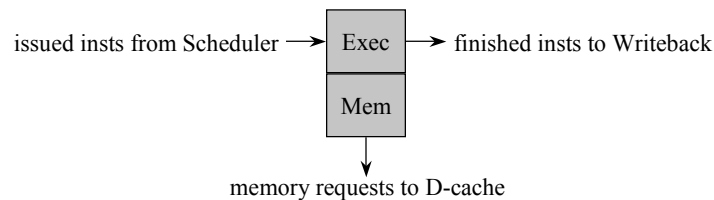
Out-of-Order Issue Simulator: Scheduler



- 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

SimpleScalar
LLC

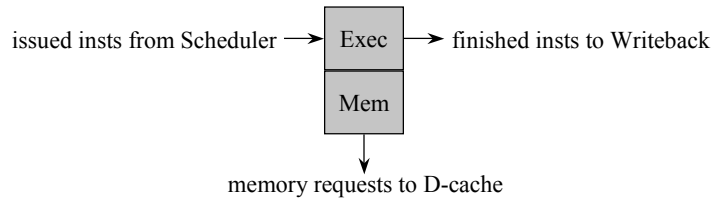
Out-of-Order Issue Simulator: Execute



- 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

SimpleScalar
LLC

Out-of-Order Issue Simulator: Execute



- 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

**SimpleScalar
LLC**

Resource Manager (resource.[hcl])

```

/* resource descriptor */
struct res_desc {
    char *name; /* name of functional unit */
    int quantity; /* total instances of this unit */
    int busy; /* non-zero if this unit is busy */
    struct res_template {
        int class; /* matching resource class */
        int oplat; /* operation latency */
        int issuelat; /* issue latency */
    } x[MAX_RES_CLASSES];
};

/* create a resource pool */
struct res_pool *res_create_pool(char *name, struct res_desc *pool, int ndesc);

/* get a free resource from resource pool POOL */
struct res_template *res_get(struct res_pool *pool, int class);
  
```

- generic resource manager
 - handles most any resource, e.g., ports, fn units, buses, etc...
 - manager maintains resource availability
 - configure with a resource descriptor list
 - busy = cycles until available

**SimpleScalar
LLC**

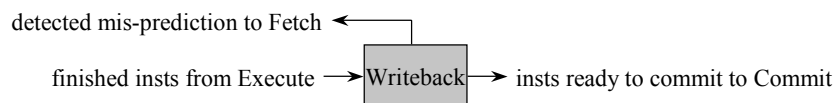
Resource Manager (resource.hcl)

```
/* resource pool configuration */
struct res_desc fu_config[] = {
  { "integer-ALU", 4, 0,
    { { IntALU, 1, 1 } } },
  { "integer-MULT/DIV", 1, 0,
    { { IntMULT, 3, 1 }, { IntDIV, 20, 19 } } },
  { "memory-port", 2, 0,
    { { RdPort, 1, 1 }, { WrPort, 1, 1 } } }
};
```

- resource pool configuration:
 - instantiate with configuration descriptor list
 - i.e., { "name", num, { FU_class, issue_lat, op_lat }, ... }
 - one entry per "type" of resource
 - class IDs indicate services provided by resource instance
 - multiple resource "types" can service same class ID
 - earlier entries in list given higher priority

SimpleScalar
LLC

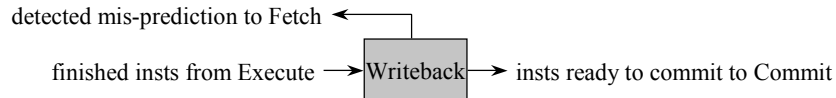
Out-of-Order Issue Simulator: Writeback



- 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

SimpleScalar
LLC

Out-of-Order Issue Simulator: Writeback



- 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 inst are satisfied, wake it up (memory dependence check occurs later in Issue)

SimpleScalar
LLC

Optimization: Fast Functional State Recovery

```

/* speculation mode, non-zero when mis-speculating */
static int spec_mode = FALSE;

/* integer register file */
static BITMAP_TYPE(SS_NUM_REGS, use_spec_R);
static SS_WORD_TYPE spec_regs_R[SS_NUM_REGS];

/* general purpose register accessors */
#define GPR(N) (BITMAP_SET_P(use_spec_R, R_BMAP_SZ, (N)) \
? spec_regs_R[N] \
: regs_R[N])

#define SET_GPR(N,EXPR) (spec_mode \
? (spec_regs_R[N] = (EXPR), \
  BITMAP_SET(use_spec_R, R_BMAP_SZ, (N)), \
  spec_regs_R[N]) \
: (regs_R[N] = (EXPR)))

/* reset copied-on-write register bitmasks back to non-speculative state */
BITMAP_CLEAR_MAP(use_spec_R, R_BMAP_SZ);

/* speculative memory hash table */
static struct spec_mem_ent *store_hstable[STORE_HASH_SIZE];
    
```

- early execution permits early detection of mispeculation
 - when misspeculation begins, all new state definitions redirected
 - copy-on-write bits indicate speculative defs, reset on recovery
 - speculative memory defs in store hash table, flushed on recovery

SimpleScalar
LLC

Out-of-Order Issue Simulator: Commit

insts ready to commit from Writeback → 

- 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

SimpleScalar
LLC

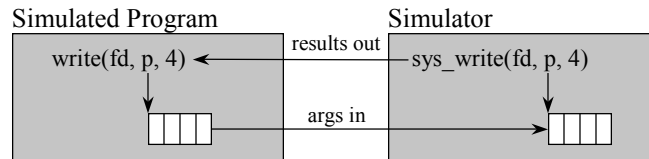
Out-of-Order Issue Simulator: Commit

insts ready to commit from Writeback → 

- 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

SimpleScalar
LLC

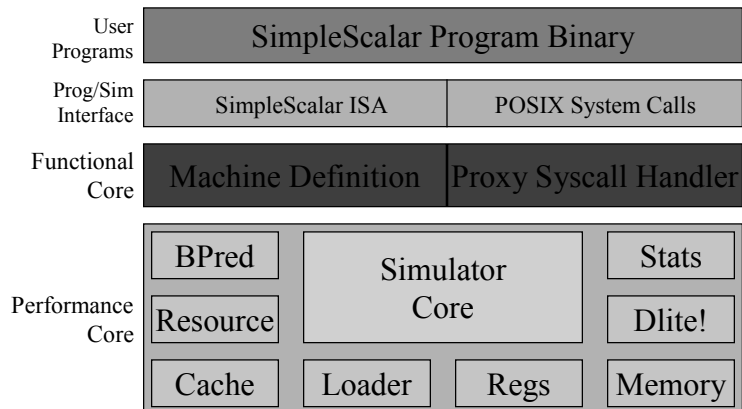
System I/O



- 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

SimpleScalar
LLC

Simulator Structure



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

SimpleScalar
LLC

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 -dumpconfig)
 - config <file> - read configuration parameters from <file>
 - dumpconfig <file> - save configuration parameters into <file>
- configuration files
 - to generate a configuration file
 - specify non-default options on command line
 - and, include “-dumpconfig <file>” to generate configuration file
 - comments allowed in configuration files
 - text after “#” ignored until end of line
 - reload configuration files using “-config <file>”
 - config files may reference other configuration files

SimpleScalar
LLC

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:mlat <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:dl1 <config> - level 1 data cache configuration
 - cache:dl1lat <cycles> - level 1 data cache hit latency

SimpleScalar
LLC

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

SimpleScalar
LLC

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

SimpleScalar
LLC

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 bimodal predictors (when “`-bpred bimod`” is specified)

`-bpred:bimod <size>` size of direct-mapped BTB

SimpleScalar
LLC

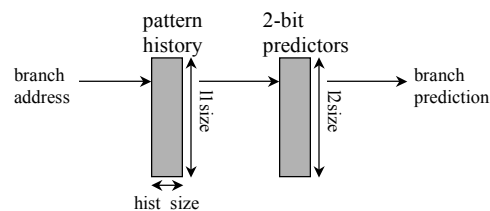
Specifying the Branch Predictor (cont.)

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

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

where

<code><l1size></code>	size of the first level table
<code><l2size></code>	size of the second level table
<code><hist_size></code>	history (pattern) width



SimpleScalar
LLC

Multi-level Cache Simulator

- Options 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
-

SimpleScalar
LLC

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

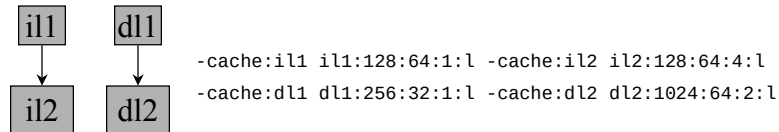
il1:1024:32:2:l 2-way set-assoc 64k-byte cache, LRU

dtlb:1:4096:64:r 64-entry fully assoc TLB w/ 4k pages,
random replacement

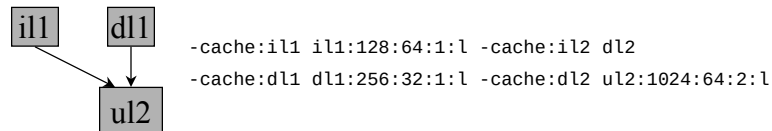
SimpleScalar
LLC

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



SimpleScalar
LLC

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

```

-ptrace F00.trc :           - trace entire execution to F00.trc
-ptrace BAR.trc 100:5000    - trace from inst 100 to 5000
-ptrace UXXE.trc :10000     - trace until instruction 10000
  
```

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

SimpleScalar
LLC

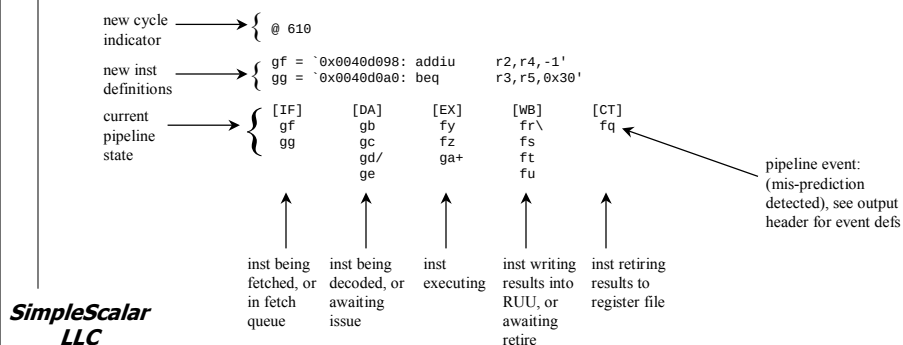
```
pipeview.pl <ptrace_file>
```

Sim-Outorder Pipetraces (cont.)

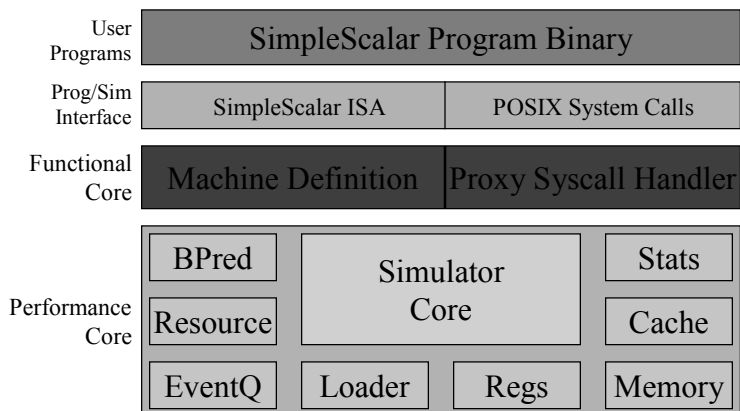
- example usage

```
sim-outorder -ptrace F00.trc :1000 test-math
pipeview.pl F00.trc
```

- example output



Simulator Structure



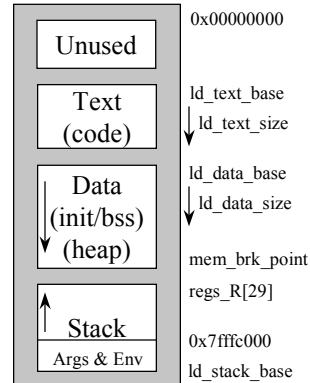
- modular components facilitate "rolling your own"
- performance core is optional

SimpleScalar LLC

Loader Module (loader.[hcl])

```
/* load program text and initialized data into simulated virtual memory
   space and initialize program segment range variables */
void ld_load_prog(mem_access_fn mem_fn, /* user-specified memory accessor */
    int argc, char **argv, /* simulated program cmd line args */
    char **envp, /* simulated program environment */
    int zero_bss_segs); /* zero uninit data segment? */
```

- prepares program memory for execution
 - loads program text section (code)
 - loads program data sections
 - initializes BSS section
 - sets up initial call stack
 - program arguments (argv)
 - user environment (envp)



SimpleScalar
LLC

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 simulator
 - generates fast and reliable codes with minimum effort
- instruction definition example

```
DEFINST(ADDI,
assembly → "addi",
template → IntALU,
FU req's → GPR(RT), NA,
output deps → SET_GPR(RT, GPR(RS)+IMM))
opcode → 0x41,
inst flags → "t, s, i",
F_ICOMP | F_IMM,
input deps → GPR(RS), NA, NA
```

Annotations in the diagram:

- assembly**: points to the instruction name "addi"
- template**: points to the functional unit IntALU
- FU req's**: points to the register requirements GPR(RT), NA
- output deps**: points to the output register SET_GPR(RT)
- opcode**: points to the instruction opcode 0x41
- inst flags**: points to the instruction flags "t, s, i"
- input deps**: points to the input registers GPR(RS), NA, NA
- semantics**: points to the instruction semantics SET_GPR(RT, GPR(RS)+IMM)

SimpleScalar
LLC

Crafting a Functional Component

```
#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,01,02,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 DEFLINK
#undef CONNECT
}
```

SimpleScalar
LLC

Crafting an Decoder

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

switch (SS_OPCODE(inst)) {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,01,02,I1,I2,I3,EXPR) \
    case OP: \
        out1 = DEP_##01; out2 = DEP_##02; \
        in1 = DEP_##I1; in2 = DEP_##I2; in3 = DEP_##I3; \
        break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT) \
    case OP: \
        /* can speculatively decode a bogus inst */ \
        op = NOP; \
        out1 = NA; out2 = NA; \
        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;
}
```

SimpleScalar
LLC

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()`

SimpleScalar
LLC

Stats Package (stats.[hc])

- one-stop module for counters, expressions, and distributions
- counters are “registered” by type with the stats package
 - see `stat_reg_*()` interfaces
 - register an expression of other stats with `stat_reg_formula()`
 - for example: `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)
 - via `stat_print_stats()` interface
- distributions also supported
 - use `stat_reg_dist()` to register an array distribution
 - use `stat_reg_sdist()` for a sparse distribution

SimpleScalar use `stat_add_sample()` to add samples
LLC

Branch Predictors (bpred.[hc])

- various branch predictors
 - static
 - BTB w/ 2-bit saturating counters
 - 2-level adaptive
- important interfaces
 - use `bpred_create(class, size)` to create a predictor
 - use `bpred_lookup(pred, br_addr)` to make a prediction
 - use `bpred_update(pred, br_addr, targ_addr, result)` to update predictions

SimpleScalar
LLC

Cache Module (cache.[hc])

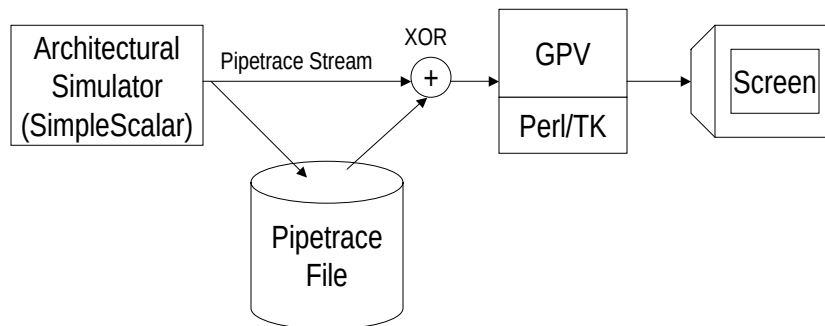
- ultra-vanilla cache module
 - can implement low- and high-associative caches, TLBs, etc...
 - efficient for all cache geometries
 - assumes a single-ported, fully pipelined backside bus
- important interfaces
 - use `cache_create(name, nsets, bsize, balloc, usize, assoc, repl, blk_fn, hit_latency)` to create a cache instance
 - use `cache_access(cache, op, addr, ptr, nbytes, when, udata)` to access a cache instance
 - use `cache_probe(cache, addr)` to check for a hit/miss without accessing the cache
 - use `cache_flush(cache, when)` to flush a cache of all contents
 - use `cache_flush_addr(cache, addr, when)` to flush a block

SimpleScalar
LLC

Additional Tools Provided with SimpleScalar

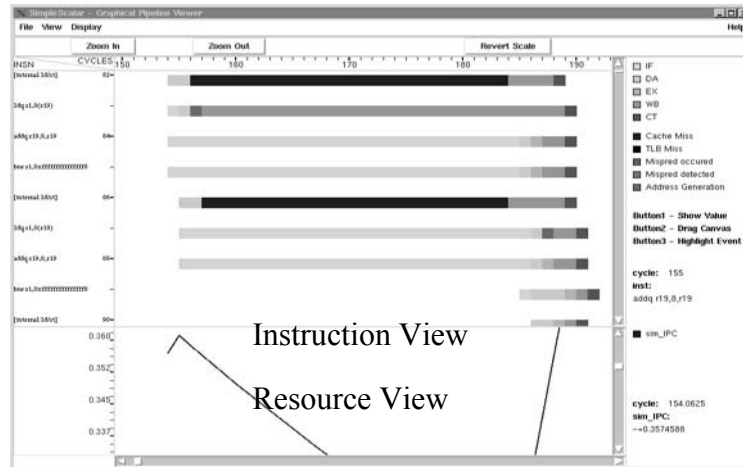
SimpleScalar
LLC

GPV Software Architecture



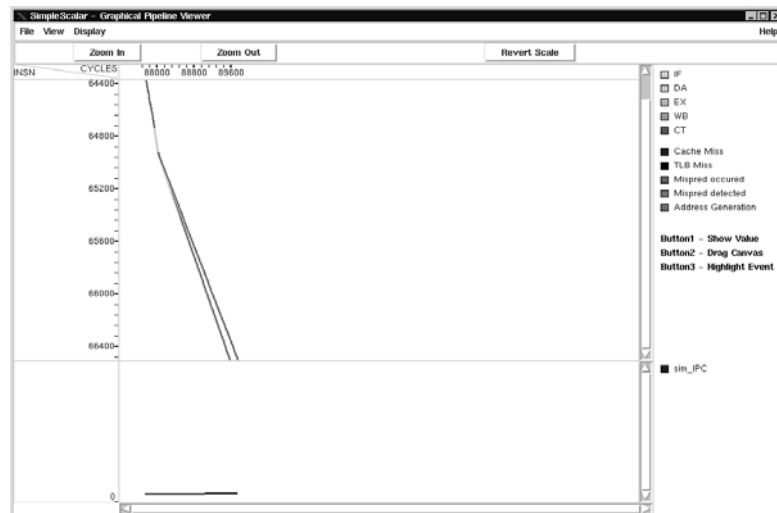
SimpleScalar
LLC

Main Window



SimpleScalar
LLC

Zoom Feature



SimpleScalar
LLC

Zoom Feature



SimpleScalar
LLC

Pipetrace Format

The @ sign marks a start of a new simulation cycle
The - sign marks the removal of an instruction
The * sign indicates a change in the instruction status

```
@ 154
* 61 CT 0x000 0 0x000
- 61
* 72 WB 0x000 0 0x000
* 71 WB 0x000 0 0x000
* 74 EX 0x001 30 0x001
* 75 EX 0x010 30 0x001
* 76 EX 0x000 0 0x001
+ 82 0x12002e558 0x00000000 [internal ld/st]
* 82 DA 0x000 0 0x000
* 79 DA 0x000 0 0x000
* 80 DA 0x000 0 0x000
* 81 DA 0x000 0 0x000
....more lines....
<sim_num_insn> 55
<sim_cycle> 154
<sim_IPC> 0.3571

@ 155
* 76 WB 0x000 0 0x000
* 75 WB 0x000 0 0x000
* 78 EX 0x001 29 0x001
* 79 EX 0x010 29 0x001
* 80 EX 0x000 0 0x001
+ 86 0x12002e558 0x00000000 [internal ld/st]
* 86 DA 0x000 0 0x000
* 83 DA 0x000 0 0x000
+ 87 0x12002e558 0x00000000 ldq r1,0(r19)
* 87 IF 0x000 0 0x001
+ 88 0x12002e55c 0x00000000 addq r19,8,r19
* 88 IF 0x000 0 0x001
<sim_num_insn> 56
<sim_cycle> 155
<sim_IPC> 0.3613

<END VISUAL>
```

Variables that the user want to track at in <> with the value

The + sign indicates a new instruction

SimpleScalar
LLC

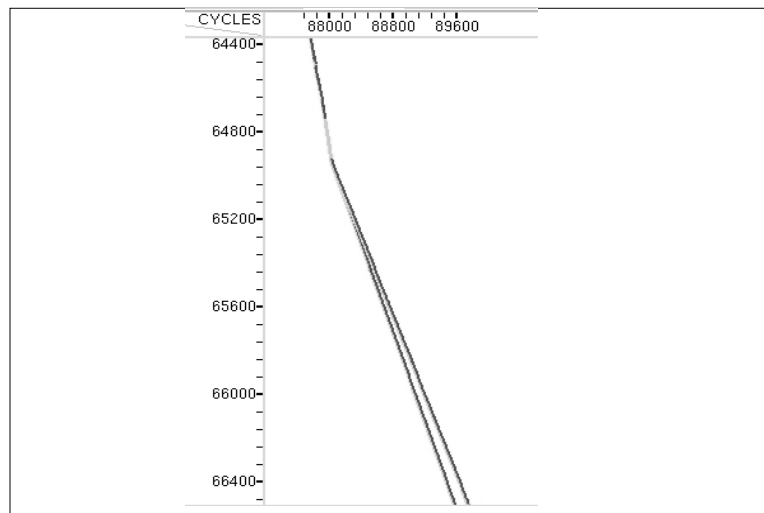
Sample H/W Optimization Add a Multiplier

- RC6 does back to back multiplies per iteration
- 4 cycles per multiply on SA-110
- Add Second Multiplier and reschedule code
- 30% speed improvement using optimization

```
for (ii=38; ii >= 4; ii-=2)
{
    x = (D+D+1);
    w = (B+B+1);
    t = x*D;
    u = w*B;
    t = CONST_ROTLL(t, 5);
    u = CONST_ROTLL(u, 5);
    C -= S[ii];
    A -= S[ii+1];
    C = ROTR(C, u)^t;
    A = ROTR(A, t)^u;
    if (ii==4)
    { tmp = A; A = B; B = C; C = D; D = tmp;
    }
    else
    { tmp = A; A = D; D = C; C = B; B = tmp;
    }
}
```

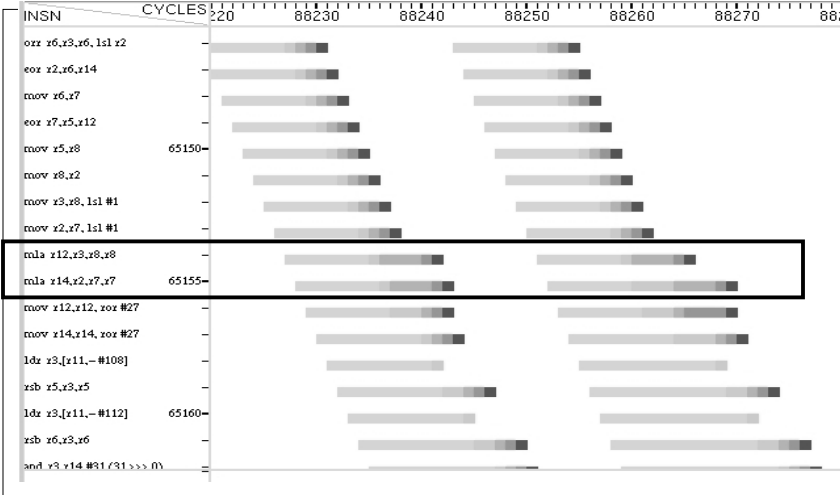
SimpleScalar
LLC

Multiplier Optimization



SimpleScalar
LLC

Multiplier Optimization (zoom)



**SimpleScalar
LLC**

GPV: Graphical Pipeline Viewer

- Portable pipeline visualization infrastructure
 - Developed by Chris Weaver, Kenneth Barr, Eric Marsman, Dan Ernst
- Provide visual platform for locating bottlenecks
 - Pipetrace view displays program slowdowns
- Enable visual diagnosis of bottleneck causes
 - Color-coded latencies identify problem delays
 - Resource view reveals resource bottlenecks
- Permit visual evaluation of program/design updates
 - Multiple trace comparisons
- Allow use on multiple platforms with multiple simulators
 - Portable code in Perl/TK
 - Standard pipetrace input

**SimpleScalar
LLC**

DLite!, the Lite Debugger

- a 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/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

SimpleScalar
LLC

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

SimpleScalar
LLC

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)

SimpleScalar
LLC

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”

SimpleScalar
LLC

PC-Based Statistical Profiles (-pcstat)

- produces text segment profile for any integer statistical counter
- supported on sim-cache, sim-profile, and sim-outorder
- specify 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>
```

SimpleScalar
LLC

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

```
executed
13 times → { 00401a10: ( 13,  0.01): <strtod+220> addiu $a1[5],$zero[0],1
               strtod.c:79
             { 00401a18: ( 13,  0.01): <strtod+228> bc1f 00401a30 <strtod+240>
               strtod.c:87
             }
never      { 00401a20:          : <strtod+230> addiu $s1[17],$s1[17],1
executed   { 00401a28:          : <strtod+238> j 00401a58 <strtod+268>
             strtod.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 including those added by users!

SimpleScalar
LLC

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 UM
 - 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-associative 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-associative analyses
 - M <size> - maximum cache size of interest
 - c <size> - cache size for direct-mapped analyses

SimpleScalar
LLC