SimpleScalar Overview

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

A Computer Architecture Simulator Primer

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

  Device
  Simulator

  System Inputs ➔ System Outputs
  ➔ System Metrics

• 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 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

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
The Zen of Hardware Model Design

Performance

Design Space

Flexibility

Detail

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

SimpleScalar LLC

A Taxonomy of Hardware Modeling Tools

Hardware Models

Architectural

Trace-Driven Exec-Driven

Emulation

Micro-Architectural

Scheduler Cycle Timers H/W Monitor

Direct Execution

- Shaded tools are included in the SimpleScalar tool set

SimpleScalar LLC
**Functional vs. Performance Simulators**

- **Functional Simulators** implement the architecture
  - The architecture is what programmers 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 instructions 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
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/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
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

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
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

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SimpleScalar</th>
<th>SA-1110</th>
<th>% Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_hit</td>
<td>1.02</td>
<td>1.01</td>
<td>0.9</td>
</tr>
<tr>
<td>cache_miss</td>
<td>33.87</td>
<td>33.70</td>
<td>0.5</td>
</tr>
<tr>
<td>br_taken</td>
<td>1.04</td>
<td>1.02</td>
<td>1.9</td>
</tr>
<tr>
<td>br_nottaken</td>
<td>1.97</td>
<td>1.91</td>
<td>3.1</td>
</tr>
<tr>
<td>bzip2 10</td>
<td>3.20</td>
<td>3.10</td>
<td>3.2</td>
</tr>
<tr>
<td>ecc1 -O cccin.i</td>
<td>2.84</td>
<td>2.90</td>
<td>2.1</td>
</tr>
<tr>
<td>fft short.pcm</td>
<td>1.45</td>
<td>1.44</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Sample Software Optimization: Loop Unrolling

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

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

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

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.
### Benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>jpeg enc/dec</td>
<td>ghostscript</td>
<td>dijkstra</td>
<td>blowfish enc/dec</td>
<td>CRC32</td>
</tr>
<tr>
<td>bitcount</td>
<td>lame</td>
<td>ispell</td>
<td>patricia</td>
<td>pgp sign</td>
<td>FFT</td>
</tr>
<tr>
<td>qsort</td>
<td>mad</td>
<td>raythn</td>
<td>(CRC32)</td>
<td>pgp verify</td>
<td>IFFT</td>
</tr>
<tr>
<td>susan (edges)</td>
<td>tiff2bw</td>
<td>sphinx</td>
<td>(sha)</td>
<td>rijndael enc/dec</td>
<td>ADPCM enc/dec</td>
</tr>
<tr>
<td>susan (corners)</td>
<td>tiff2rgba</td>
<td>stringsearch</td>
<td>(blowfish)</td>
<td>sha</td>
<td>GSM enc/dec</td>
</tr>
<tr>
<td>susan (smoothing)</td>
<td>tiffdither</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tiffmedian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Distribution

![Instruction Distribution Chart](chart.png)
ARM Configurations

<table>
<thead>
<tr>
<th>Feature</th>
<th>SA-1100</th>
<th>Xscale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch queue (instructions)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Not-taken</td>
<td>8k bimodal, 2k 4-way BTB</td>
</tr>
<tr>
<td>Fetch &amp; Decode width</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Functional Units</td>
<td>1 int ALU, 1 FP mult, 1 FP ALU</td>
<td>1 int ALU, 1 FP mult, 1 FP ALU</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>16k, 32-way</td>
<td>32k, 32-way</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>16k, 32-way</td>
<td>32k, 32-way</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>4-byte</td>
<td>4-byte</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>12 cycle</td>
<td>12 cycle</td>
</tr>
</tbody>
</table>

Achieved IPC

Graph showing IPC values for various benchmarks on SA-1100 and Xscale processors.
Simulation Suite Overview

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Lines</th>
<th>Functionality</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim-Fast</td>
<td>420</td>
<td>functional</td>
<td>- 420 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 4+ MIPS</td>
<td></td>
</tr>
<tr>
<td>Sim-Safe</td>
<td>350</td>
<td>functional w/ checks</td>
<td></td>
</tr>
<tr>
<td>Sim-Profile</td>
<td>900</td>
<td>functional, lot of stats</td>
<td></td>
</tr>
<tr>
<td>Sim-Cache/</td>
<td>&lt; 1000</td>
<td>functional, cache stats</td>
<td></td>
</tr>
<tr>
<td>Sim-Cheetah</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sim-Outorder</td>
<td>3900</td>
<td>performance</td>
<td>- 3900 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- OoO issue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- branch pred.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- mis-spec.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ALUs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- TLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- 200+ KIPS</td>
</tr>
</tbody>
</table>

Performance

Detail

Simulator Structure

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

Fetch → Dispatch → Scheduler → Exec → Writeback → Commit

I-Cache (IL1) → I-TLB → Virtual Memory → I-Cache (IL2)

D-Cache (DL1) → D-TLB → Virtual Memory → D-Cache (DL2)

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

Out-of-Order Issue Simulator: Main

```c
ruu_init()
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    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
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

Procedure (once per cycle)
- fetch instructions 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
  - speeds up memory dependence checking
**SimpleScalar LLC**

The Register Update Unit (RUU)

- RUU handles register synchronization/communication
  - unifies reorder buffer and reservation stations
  - managed as a circular queue
  - entries allocated at Dispatch, deallocated at Commit
  - out-of-order issue, when register and memory deps satisfied
  - memory dependencies resolved by load/store queue (LSQ)

### Optimization: Output Dependence Chains

```c
/* a reservation station link: this structure links elements of a RUU
 reservation station list; used for ready instruction queue, event queue, and
 output dependency lists; each RS_LINK node contains a pointer to the RUU
 entry it references along with an instance tag, the RS_LINK is only valid if
 the instruction instance tag matches the instruction RUU entry instance tag;
 this strategy allows entries in the RUU can be squashed and reused without
 updating the lists that point to it, which significantly improves the
 performance of (all to frequent) squash events */
struct RS_link {
  struct RS_link *next;       /* next entry in list */
  struct RUU_station *rs;     /* referenced RUU resv station */
  INST_TAG_TYPE tag;          /* inst instance sequence number */
  union {
    SS_TIME_TYPE when;       /* time stamp of entry (for eventq) */
    INST_SEQ_TYPE seq;       /* inst sequence */
    int opnum;               /* input/output operand number */
  } x;
};
```

- register dependencies described with dependence chains
  - rooted in RUU of defining instruction, one per output register
  - also rooted in create vector, at index of logical register
- output dependence chains walked during Writeback
  - same links used for event queue, ready queue, etc...

**SimpleScalar LLC**
**Optimization: Output Dependence Chains**

```c
static INLINE void ruu_link_idep(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, 
           indicates 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;
}
```

**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

```c
/* 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->rs->idep_ready[olink->x.opnum] = TRUE;
    }
    /* grab link to next element prior to free */
    olink_next = olink->next;
}
```
The Load/Store Queue (LSQ)

- LSQ handles memory synchronization/communication
  - contains all loads and stores in program order
    - load/store primitives really, address calculation is separate op
    - effective address calculations reside in RUU (as ADD insts)
  - loads issue out-of-order, when memory deps known satisfied
    - load addr known, source data identified, no unknown store address

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

Resource Manager (resource.hc)

/* 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 issuclat; /* 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)

```c
/* 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

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

Optimization: Fast Functional State Recovery

```c
/* 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_htable[STORE_HASH_SIZE];
```

- early execution permits early detection of misspeculation
  - 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
Out-of-Order Issue Simulator: 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

SimpleScalar
LLC

Out-of-Order Issue Simulator: 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
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

Simulator Structure

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

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
  -decoder: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:dlllat <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 bimodal predictors (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

Diagram:

- Branch address
- Branch history
- Pattern width
- 2-bit predictors
- Branch prediction
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

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
    - 1 - for LRU
    - r - for FIFO
    - r - for RANDOM

- examples

  il1:1024:32:2:1 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 il1:128:64:1:l -il2 dl2
```

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 FOO.trc : - trace entire execution to FOO.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

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

- example usage

```plaintext
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'

[IF]      [DA]      [EX]      [WB]      [CT]
gf        gb        fy        fr        fq
          gc        fz        fs
          gd        ga+       ft
              ge                  fu
```

pipeline event: mis-prediction detected, see output header for event defs

SimpleScalar LLC

Simulator Structure

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

SimpleScalar LLC
Loader Module (loader.hc)

- 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)

```c
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 */
```

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, 0x41, "addi", "t,s,i", IntALU, F_ICOMP | F_IMM, GPR(RT), NA, NA)
SET_GPR(RT, GPR(RS)+IMM)
```

```plaintext
Opcode: 0x41
Assembly: addi
Template: t,s,i
Instruction Flags: F_ICOMP | F_IMM
Functional Unit: IntALU
Input Deps: GPR(RS), NA
Output Deps: 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)
    #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;
    #define CONNECT(OP)
    #include "ss.def"
    #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:                                                  
            /* 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;
    }
```
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 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
  - use stat_add_sample() to add samples
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

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
Additional Tools Provided with SimpleScalar

GPV Software Architecture
**SimpleScalar LLC**

**Zoom Feature**

![Zoom Feature Diagram]

**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.

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

The + sign indicates a new instruction.

---

SimpleScalar LLC

---

37
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_ROTL(t, 5);
    w = CONST_ROTL(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;
    }
}
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
DLite!, the Lite Debugger

- a lightweight symbolic debugger
  - supported by all simulators (except sim-fast)
- designed for easy 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

DLite! expressions

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

DLite!, the Lite Debugger (cont.)

- breakpoints
  - code
    - break <addr>
    - e.g., break main, break 0x600148
  - 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-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)
  - -ampprof - 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 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_hits.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

  executed 13 times

<table>
<thead>
<tr>
<th>Address</th>
<th>Frequency</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040001a0</td>
<td>13, 0.01</td>
<td>&lt;strtod+220&gt; addiu $a1[5],$zero[0],1</td>
</tr>
<tr>
<td>0040001a18</td>
<td>13, 0.01</td>
<td>&lt;strtod+228&gt; beqz 00401a30 &lt;strtod+240&gt;</td>
</tr>
<tr>
<td>0040001a20</td>
<td></td>
<td>&lt;strtod+230&gt; addiu $a1[17],$a1[17],1</td>
</tr>
<tr>
<td>0040001a28</td>
<td></td>
<td>&lt;strtod+238&gt; j 00401a58 &lt;strtod+268&gt;</td>
</tr>
<tr>
<td>0040001a30</td>
<td>13, 0.01</td>
<td>&lt;strtod+240&gt; mul.d $f2,$f20,$f4</td>
</tr>
<tr>
<td>0040001a38</td>
<td>13, 0.01</td>
<td>&lt;strtod+248&gt; addiu $v0[2],$v0[13],-48</td>
</tr>
<tr>
<td>0040001a40</td>
<td>13, 0.01</td>
<td>&lt;strtod+250&gt; mcall $v0[2],$f0</td>
</tr>
</tbody>
</table>

- works on any integer counter including those added by users!
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
  - \texttt{-refs \{inst, data, unified\}} - specify reference stream to analyze
  - \texttt{-C \{fa, sa, dm\}} - cache config, i.e., fully or set-assoc or direct
  - \texttt{-R \{lru, opt\}} - replacement policy
  - \texttt{-a <sets>} - log base 2 number of set in minimum config
  - \texttt{-b <sets>} - log base 2 number of set in maximum config
  - \texttt{-l <line>} - cache line size in bytes
  - \texttt{-n <assoc>} - maximum associativity to analyze (log base 2)
  - \texttt{-in <interval>} - cache size interval for fully-assoc analyses
  - \texttt{-M <size>} - maximum cache size of interest
  - \texttt{-c <size>} - cache size for direct-mapped analyses