Processor Simulation

- Simulation is essential to processor research and development

- Micro-architecture simulation is getting harder
  - Micro-architecture complexity is increasing
  - Acceptable benchmark workloads are getting larger
Out-Of-Order Processor Simulation

- State of the art out-of-order simulators:
  - SimpleScalar — 4,000 times slowdown
  - RSIM — 15,000 insts/sec on a SUN Ultra 1/140
  - SimOS/MXS — several thousand times slowdown

- Our previous work:
  - FastSim — 170-360 times slowdown!
  - Uses fast-forwarding (i.e., run-time specialization + memoization)
  - Difficult to implement by hand
New Contributions

- The Facile simulation language
  - Simplifies implementation of processor simulators
  - Simplifies compiler analysis needed for optimization

- The Facile compiler
  - Automatically applies the fast-forwarding optimization
Outline

- Overview of fast-forwarding
- The advantages of Facile
- Compiling Facile to use fast-forwarding
- Performance results
Repeated Simulation Work

- Processor simulators encounter the same instructions many times

- Even detailed μ-architecture state repeats often
  - E.g., in FastSim out-of-order pipeline state is repeated >99% of the time simulating the Spec95 benchmarks

- Fast-forwarding skips repeated simulation work
  - Specialize w.r.t. subset of micro-architecture state
  - Cache and re-use specialized code
Fast-Forwarding Simulator

- **rt-static micro-arch. state**
- **lookup rt-static input**
- **slow/complete simulator step function**
- **specialized action cache**
- **fast/residual step function**

Dynamic micro-arch. state
Simulator Step Function

- Our simulators are written as an outer loop around a single function—the *simulator step function*
  - A step function may simulate 1 instruction, 1 cycle, several instructions and cycles, etc.

- Cache specialized versions of the step function
  - The size of the step function influences the number of opportunities to use specialized code
Differences From Traditional Specialization

- Millions of specialized step functions generated
  - It is essential that specialized code be small

- Only executed control-flow paths are specialized
  - Saves space by not storing code for un-used paths
  - Specialized code may fail

- The specialized action cache stores interpreted action numbers, not binary code
  - One action number represents several operations
Cached Actions

Keys (run-time static input)

Dynamic Actions

Missing Control Paths
Cache Miss Recovery

- Specialized code may fail
  - E.g., dynamic condition may evaluate to false, where every previous occurrence evaluated to true

- To recover:
  - Find last run-time static input to the step function
  - Restart slow simulator using run-time static input
  - Only allow run-time static code to execute
  - When simulation reaches point of failure, return to normal slow simulation
The Facile Programming Language

- Structures simulators into step functions
  - Implicit outer loop
  - Function parameters are run-time static, global variables are dynamic

- No recursion and no pointers
  - Simplifies compiler analysis and optimization

- Plus syntax to support processor simulation
val \( R = \text{array}(32)\{0\}; \)

val init = system?start_pc;

fun main(pc) {
    val npc = pc + 4;
    switch(pc) {
        pat add:
            if(i) \( R[rd] = R[rs1] + \text{imm?sext}(32) \)
            else \( R[rd] = R[rs1] + R[rs2] \);
        pat bz:
            if(R[rd] == 0)
                npc = pc + \text{offset?sext}(32);
            npc = pc + offset?sext(32);
    }
    init = npc;
}

register file ‘\( R \)’ is a dynamic input

‘init’ holds next arguments to ‘main’

simulator step function

arguments to ‘main’ are run-time static

prepare for next call to ‘main’
fun main(pc)
{
    val npc = pc + 4;
    switch(pc) {
        pat add:
            if(i) \text{R[rd]} = \text{R[rs1]} + \text{imm?sext(32)};
            else \text{R[rd]} = \text{R[rs1]} + \text{R[rs2]};
        pat bz:
            if(\text{R[rd]} == 0)
                npc = pc + offset?sext(32);
    }
    \text{init} = npc;
}
Dynamic Basic Blocks

entry

\[ b_1: \; R[s] = R[s] + s \]

\[ b_2: \; R[s] = R[s] + R[s] \]

\[ b_3: \; R[s] == 0 \]

\[ b_4: \; \text{init} = s \]

exit
fun fast_main(){
    while(true) {
        switch(get_next_action_number()) {
            case INDEX_ACTION:
                verify_static_input();
            case 1: read_static_data(rd, rs1, t1);
                R[rd] = R[rs1] + t1;
            case 2: read_static_data(r1, r2, r3);
                R[rd] = R[rs1] + R[rs2];
            case 3: read_static_data(rd);
                val t2 = (R[rd] == 0);
                verify_dynamic_result(t2);
            case 4: read_static_data(npc);
                init = npc;
        } } }

The Fast Simulator
fun slow_main(pc) {
    val npc = pc + 4;
    switch(pc) {
        pat add: {
            if(i) {
                memoize_action_number(1);
                val t1 = imm?sext(32);
                memoize_static_data(rd, rs1, t1);
                if(!recover) R[rd] = R[rs1] + t1;
            } else {
                memoize_action_number(2);
                memoize_static_data(rd, rs1, rs2);
                if(!recover) R[rd] = R[rs1] + R[rs2];
            }
        }
    }
}
Experimental Conditions

- Simulating an out-of-order micro-architecture
  - 4 instructions fetched per cycle
  - Approx. 32 instructions in window
  - Non-blocking data cache, up to 8 simultaneous loads

- SPEC95 benchmarks
  - run with their “test” input sets (except compress)

- SUN Ultra Enterprise E5000
  - 167MHz UltraSPARC processor
  - 2 GBytes of physical memory
Hand Coded FastSim Simulator

![Bar chart showing performance comparison between SimpleScalar, no FF, and hand coded FF for various benchmarks.](chart.png)
Compiled Facile Code

![Compiled Facile Code Diagram](image-url)
Facile vs. Hand Coded
Fast-Forwarding
Future Optimizations

- Compile time partial evaluation

- Liveness analysis
  - Currently, a number of extra statements are generated for variables that are not live

- Run-time code generation
  - Cache executable code rather than interpreted action numbers
Conclusion

- Fast-forwarding accelerates complex simulation
  - Especially out-of-order processor simulation

- Facile makes fast-forwarding more accessible to simulator implementers

- Different requirements have lead to a different implementation of run-time specialization