THE UNIVERSITY of EDINBURGH
school of informatics

PASTA

Processor Automated Synthesis by iTerative Analysis Project
http://groups.inf.ed.ac.uk/pasta/

icsa Institute for Computing Systems Architecture

EPSRC Pioneering research and skills

Ultra-Fast Functional Instruction Set Simulation

JIT Dynamic Binary Translation

HOTSPOT Detection

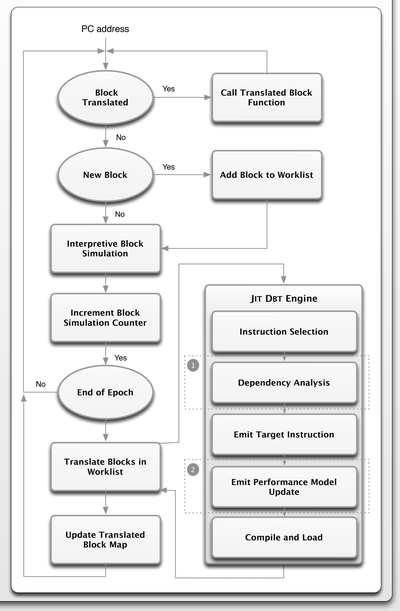
- Simulation time is partitioned into epochs
- Epoch is interval between two translations
- Frequently executed blocks are HOTSPOTS
- HOTSPOTS are translated into native code

JIT Dynamic Binary Translation

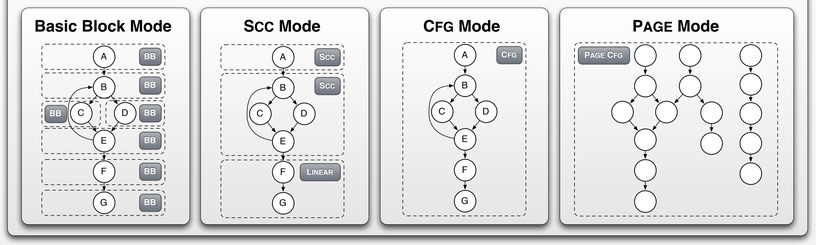
- Map ARCompact code onto native code
- Determine operand dependencies (1)
- Compute instruction side effects (1)
- Emit native target instructions
- Emit performance model updates (2)

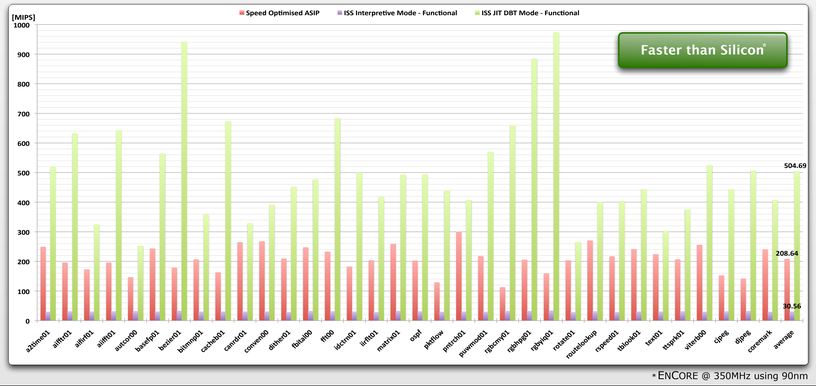
Large Translation Units

- Larger blocks result in improved locality
- More scope for JIT compiler optimisations



Large Translation Units





*ENCORE @ 350MHz using 90nm

Ultra-Fast Cycle Accurate Instruction Set Simulation

JIT Performance Model Translation

Block of ARCompact™ Instructions

```
...
0x00000848:
[0x00000848] ext   r2,r9
[0x0000084c] xor   r3,r12,r2
[0x00000850] and   r3,r3,0xf
[0x00000854] asl   r3,r3,0x3
[0x00000858] and   r2,r2,0x7
[0x0000085c] or    r3,r3,r2
[0x00000860] asl   r4,r3,0x8
[0x00000864] brcc.d  r10,r13,0x2c
[0x00000868] or    r4,r4,r3
...
```

JIT Translated Block with Performance Model

```
extern CpuState cpu;              // global processor state
void BLK_0x00000848(void) {                                   (1)
  cpu.r[2] = (uint16_t)(cpu.r[9]);
  pipeline(0,cpu.avail[9],&(cpu.avail[2]),0x00000848,1,0);
  cpu.r[3] = cpu.r[12] ^ cpu.r[2];                            (2)
  pipeline(cpu.avail[12],cpu.avail[2],&(cpu.avail[3]),0x0000084c,1,0);
  cpu.r[3] = cpu.r[3] & (uint32_t)15;
  pipeline(cpu.avail[3],0,&(cpu.avail[3]),0x00000850,1,0);
  cpu.r[3] = cpu.r[3] << ((int8_t)3 & 0x1f);
  pipeline(cpu.avail[3],0,&(cpu.avail[3]),0x00000854,1,0);
  cpu.r[2] = cpu.r[2] & (uint32_t)7;
  pipeline(cpu.avail[2],0,&(cpu.avail[2]),0x00000858,1,0);
  cpu.r[3] = cpu.r[3] | cpu.r[2];
  pipeline(cpu.avail[3],cpu.avail[2],&(cpu.avail[3]),0x0000085c,1,0);  (3)
  cpu.r[4] = cpu.r[3] << ((int8_t)8 & 0x1f);
  pipeline(cpu.avail[3],0,&(cpu.avail[4]),0x00000860,1,0);
  // compare and branch instruction with delay slot
  pipeline(cpu.avail[10],cpu.avail[13],&(ignore),0x00000864,1,0);
  if (cpu.r[10] >= cpu.r[13]) {
    cpu.pl[FE] = cpu.pl[ME] - 1; // branch penalty              (4)
    fetch(0x0000086c);           // speculative fetch due to branch pred.
    cpu.auxr[BTA] = 0x00000890;  // set BTA register
    cpu.D = 1;                   // set delay slot bit
  } else {                                                    (5)
    cpu.pc = 0x0000086c;
  }
  cpu.r[4] = cpu.r[4] | cpu.r[3];// delay slot instruction
  pipeline(cpu.avail[4],cpu.avail[3],&(cpu.avail[4]),0x00000868,1,0);
  if (cpu.D) {                   // branch was taken
    cpu.D = 0;                   // clear delay slot bit
    cpu.pc = cpu.auxr[BTA];      // set PC
  }
  cpu.cycles = cpu.pl[WB];       // set total cycle count at end of block
  return;
}
```

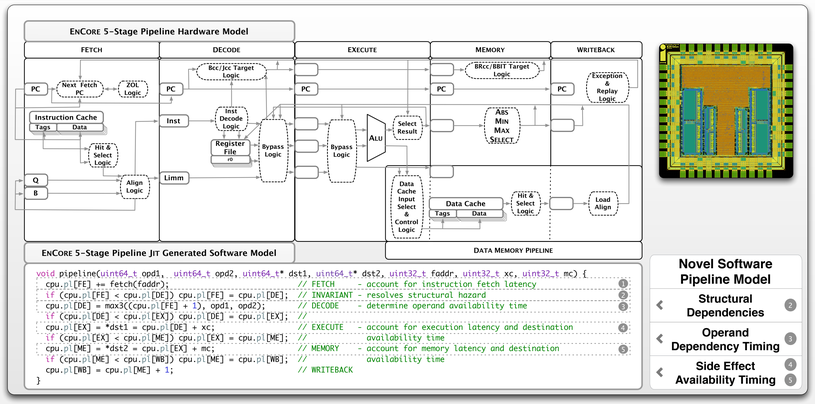
Data Structures

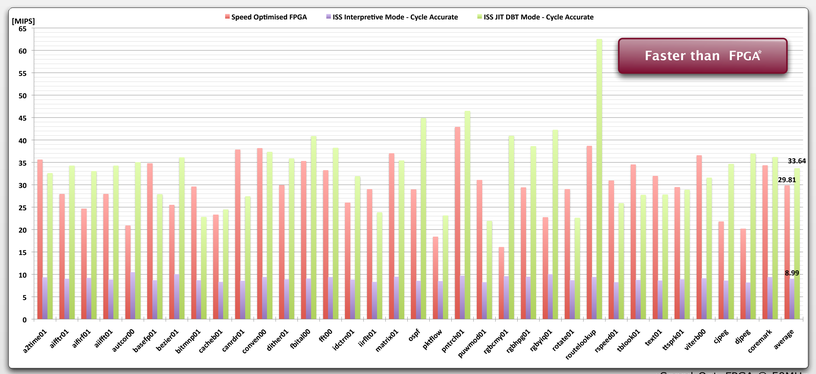
```
// pipeline stages
typedef enum {
  FE,    // fetch
  DE,    // decode
  EX,    // execute
  ME,    // memory
  WB,    // write back
  STAGES // 5 stages
} Stage;

// processor state
typedef struct {
  uint32_t pc;                                                (6)
  // general purpose registers
  uint32_t r[REGS];
  // auxiliary registers
  uint32_t auxr[AUXREGS];
  // status flags (H..halt bit)
  char     L,Z,N,C,V,U,D,H;
  // per stage cycle count                                    (7)
  uint64_t pl[STAGES];
  // per register cycle count
  uint64_t avail[REGS];
  // total cycle count
  uint64_t cycles;
  // used when insn. does not
  // produce result
  uint64_t ignore;
} CpuState;
```

High-Speed Performance Modeling

Key idea is to retain a high-speed instruction-by-instruction execution model and reconstruct microarchitectural pipeline state at instruction commit boundaries. In order to increase simulation speed we introduce an innovative pipeline software model and perform JIT dynamic binary translation. JIT generated code is augmented with highly optimised JIT generated code for performance model updates.





*Speed-Opt. FPGA @ 50MHz

ARCSIM

Flexible Faster Than Silicon Instruction Set Simulator

Full Virtual Simulation Platform

Simulation Modes

- Verified Cycle-Accurate Simulation
- High-Speed JIT DBT Simulation
- Fast Cycle-Approximate Simulation
- Co-Simulation for HW Verification

Detailed Profiling Simulation Mode

Profiling simulation mode is orthogonal to the above modes and provides detailed statistics about various simulation aspects such as dynamic instruction frequencies, per-instruction latency distributions, detailed cache-statistics, branch predictor statistics, detailed register usage statistics...

Features and APIs

Instruction Set Extension API

Powerful plugin API for implementing instruction set extensions. ISE plugins are as powerful as baseline instructions and integrate into cycle-accurate and high-speed JIT DBT simulation modes.

Memory Mapped IO API

Simple yet very powerful API for implementing user defined memory mapped IO devices such as sound, screen, ethernet...

System Call Emulation

Emulation implementations for Linux system calls as well as support for redirection of stdin, stdout, and stderr are provided and enable the standalone simulation of binaries without an operating system.

Debugging and Tracing

Interactive debugging interface supporting single step tracing of instructions, debugging of processor state, setting of breakpoints...

Instruction Set Simulator

Virtual Devices

Instruction Set Extensions