



Overview of the Edinburgh High Speed Simulator

Nigel Topham

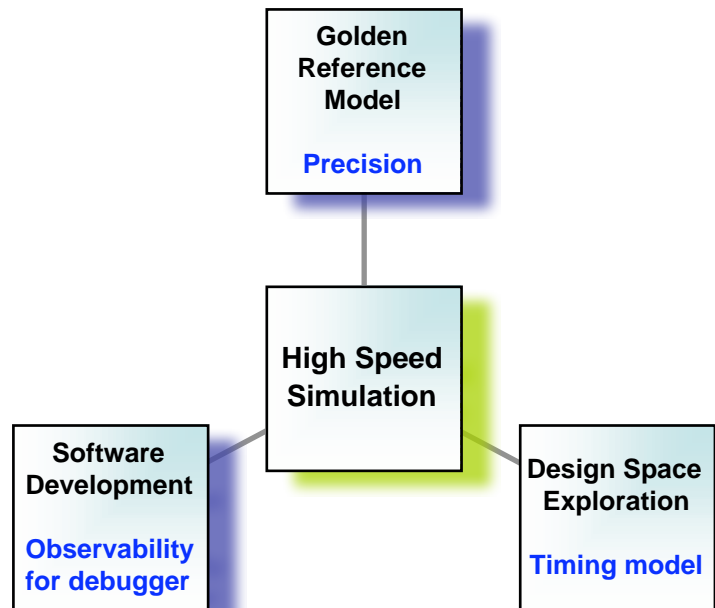
`npt@inf.ed.ac.uk`

Daniel Jones

`daniel.jones@inf.ed.ac.uk`

Institute for Computing Systems Architecture
School of Informatics
The University of Edinburgh
U.K.

Original goals of the simulator



- Design goals
 - Maximum simulation rate
 - Full system simulation
 - Precise state visibility at observation points
 - Extensible architecture
- Multi-function capability
 - Based on common simulation core
 - Configurable
- Golden reference model
 - Exact modelling of target state
 - Co-simulation API for HW verification
- Design space exploration
 - Near cycle-accurate timing models
 - Gradual reduction in speed with increasing accuracy

Critical Design Issues – Areas of Innovation

- Minimal simulation overhead for memory accesses
 - Fast translation of target virtual memory to host memory
 - Fast, precise detection of TLB misses, Protection and Alignment violations
- Engineering issues for JIT binary translation
 - Retain a precise view of target state (but elide the details)
 - Maintain coherent translations for self-modifying code
 - Persistent retention of translations between simulations
- Enable multi-core simulation
- Execution-driven performance models
 - Caches (I\$ and D\$, both configurable)
 - Pipeline timing model
 - Branch predictor
- Optional sampling - to switch between functional and cycle-accurate

Simulator Performance Benchmarks

1. Isolated instruction sequence micro-benchmarks
2. Standalone application benchmarks
3. Linux full-system simulation, focusing on dynamic JIT behaviour
4. Persistent 'learning' behaviour of JIT binary translation

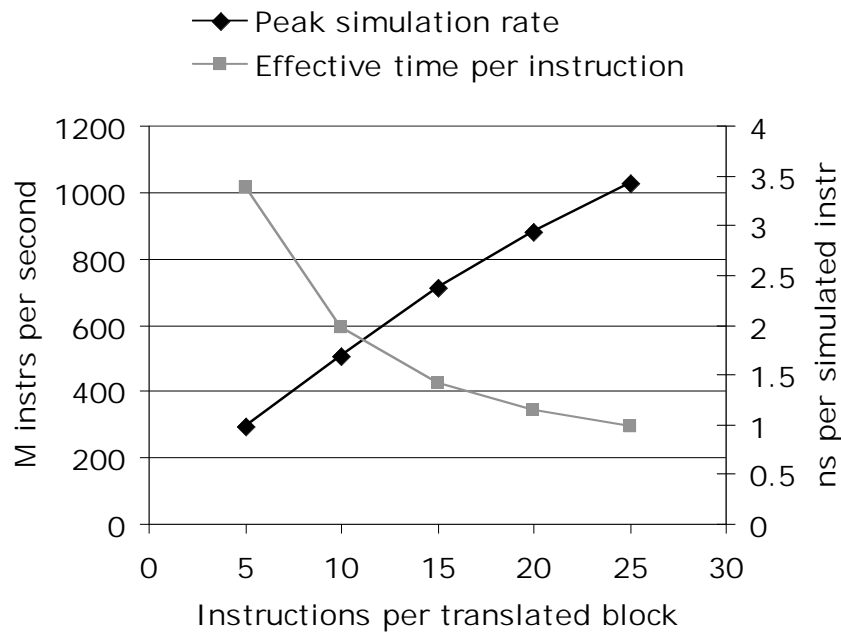
Parameter	Value
Vendor & model	Dell™ PowerEdge™ 2960
Number of CPUs	4 (2 x dual-core)
Processor	Intel® Xeon® 5160
Clock frequency	2992 MHz
L1 cache	32KB I and D caches
L2 cache	4 MB per dual-core CPU
FSB frequency	1333 MHz

Simulation Host Configuration

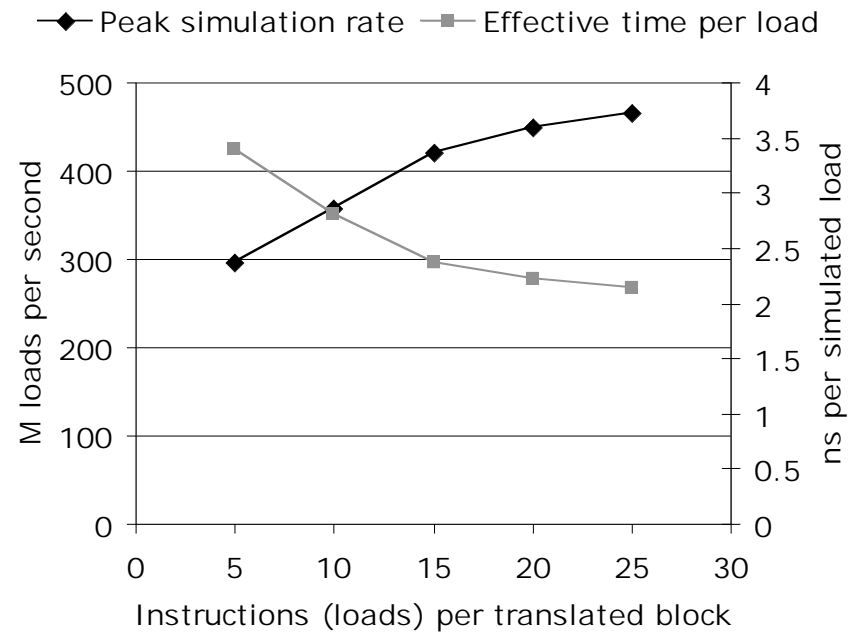
Isolated Instruction Sequence Micro-benchmarks

- JIT mode performance linked to size of translated block
 - Reduced simulation overheads
 - More scope for ILP during execution
- Investigated potential gains, through synthetic benchmarks
 - Large basic blocks = high simulation rate

ALU instructions



Load instructions



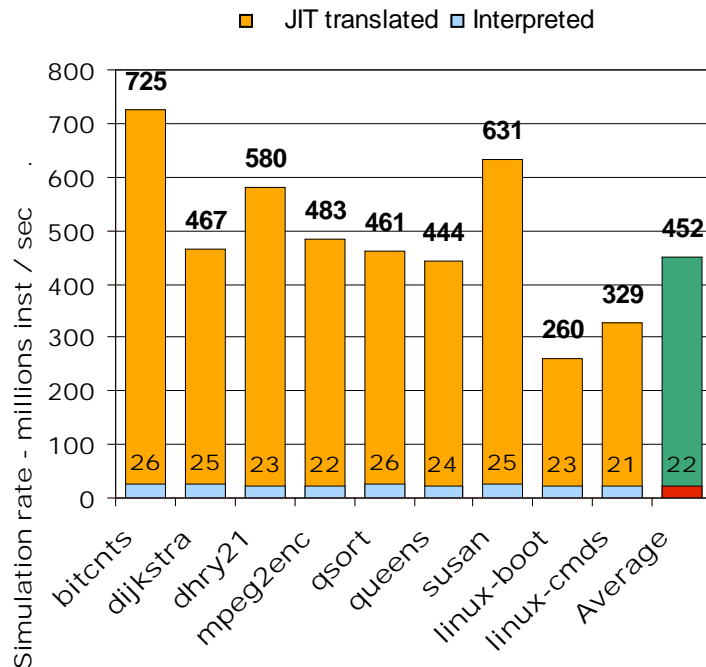
Application Benchmarks

- 7 benchmark applications, from MiBench, Mediabench etc.
- Linux boot benchmark, measured from reset to console prompt
- Decode Cache (DCC) and Block Translation Cache (BTC) measured
 - High hit ratios achieved in all but one case
 - Many small sections of code touched during Linux boot-up

Benchmark	Origin	Text size KB	Ave. Block Size		Instruction Count x 10 ⁶	Cache Hit Ratios	
			Static	Dynamic		DCC	BTC
bitcnts	MiBench	36	6.09	9.09	936	98.42	99.77
dijkstra	MiBench	39	5.74	5.26	56	98.15	99.54
dhry21	Dhrystone 2.1	34	5.85	5.62	5,280	95.14	99.99
mpegenc	Mediabench I	137	6.43	8.75	13,121	99.83	99.92
qsort	MiBench	37	5.76	7.34	137	98.73	99.74
10-queens	Traditional	27	5.94	5.89	244	98.66	99.93
susan	MiBench	60	5.71	9.47	32	97.34	98.94
linux booting	Linux 2.1.14	1,378	5.53	5.78	129	79.71	99.55

Simulator Performance Summary

Simulation Rate



N. Topham¹ and D. Jones², "High Speed CPU Simulation using JIT Binary Translation", ISCA workshop on Modeling Benchmarking and Simulation, June 2007.

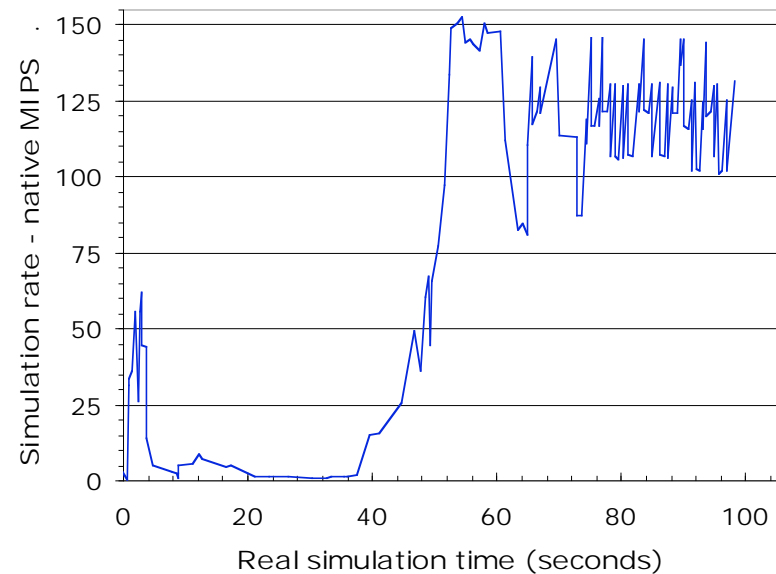
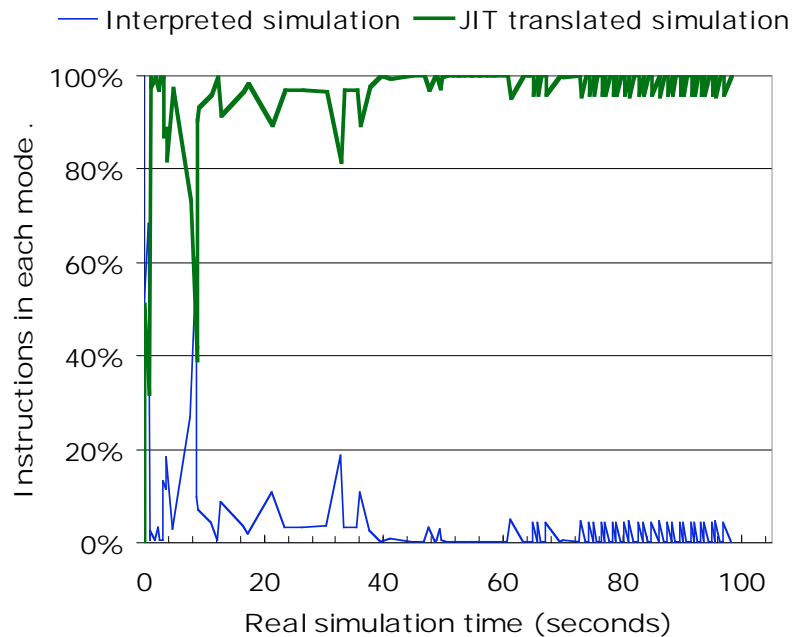
1. Work supported by EPSRC grant EP/D50399X/1

2. Supported by EPSRC PhD research studentship

- DSE requires mixed-mode high-speed simulation
 - Functional, cycle-approximate, cycle-accurate
- Purely interpretive simulation:
 - 20 - 25 MIPS (scalar operations)
 - 130 MIPS (128-bit SIMD operations)
- JIT translation yields 450 MIPS simulation rate
- More than 4x the speed of a synthesised Verilog version of the same processor running in a Xilinx Virtex-4 FPGA (100 MHz)
- Cycle-approximate model, 5 - 10 MIPS
 - Pipeline model
 - Memory hierarchy model
 - Branch prediction model
 - Power models are possible extension
- Sampling capability (functional vs cycle-approx)
 - 450 MIPS functional
 - 5 - 10 MIPS with timing models

Dynamic JIT Behaviour

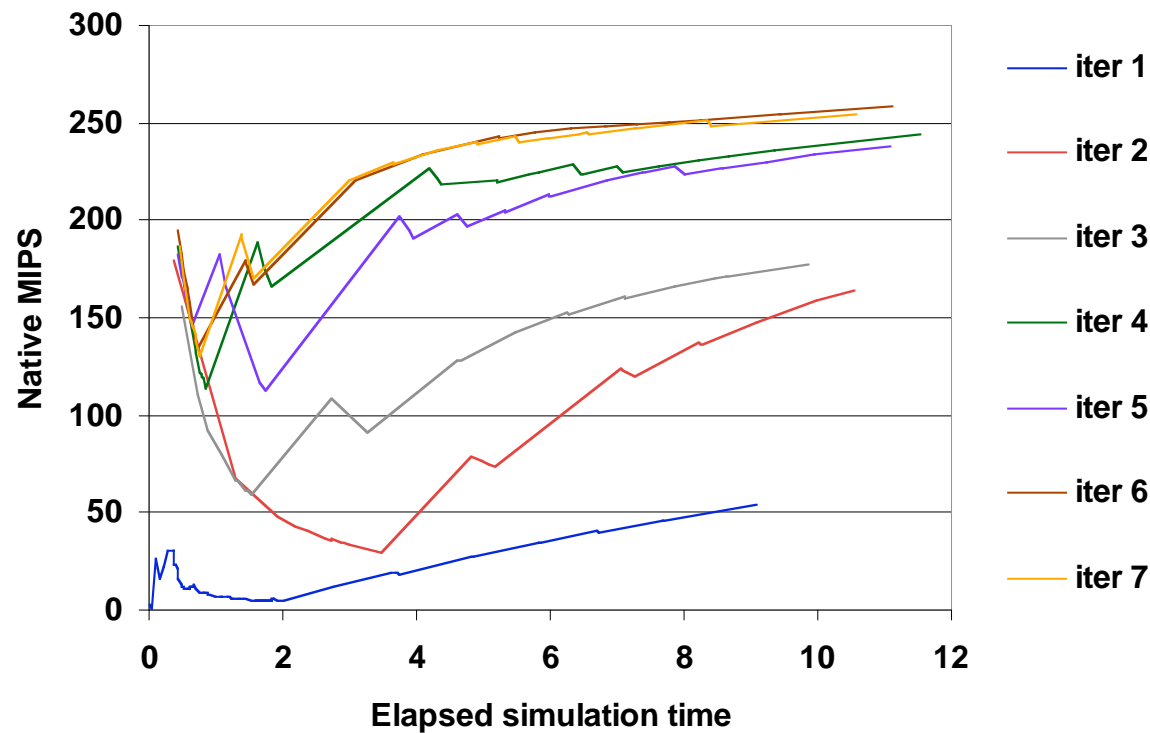
- Performance measured at each simulation epoch
 - No. instructions simulated in each mode
 - Instantaneous simulation rate (within the epoch)
- Initial ‘learning’ period, dominated by translation activity
- Speed rises rapidly after ‘hot’ regions of kernel have been translated



Persistent JIT Translation Reuse

- Iterative simulation of Linux booting, repeated 7 times
- Kept the set of translations from one run to the next
- Stabilises after 6 iterations

Cumulative MIPS - Iterative Translation



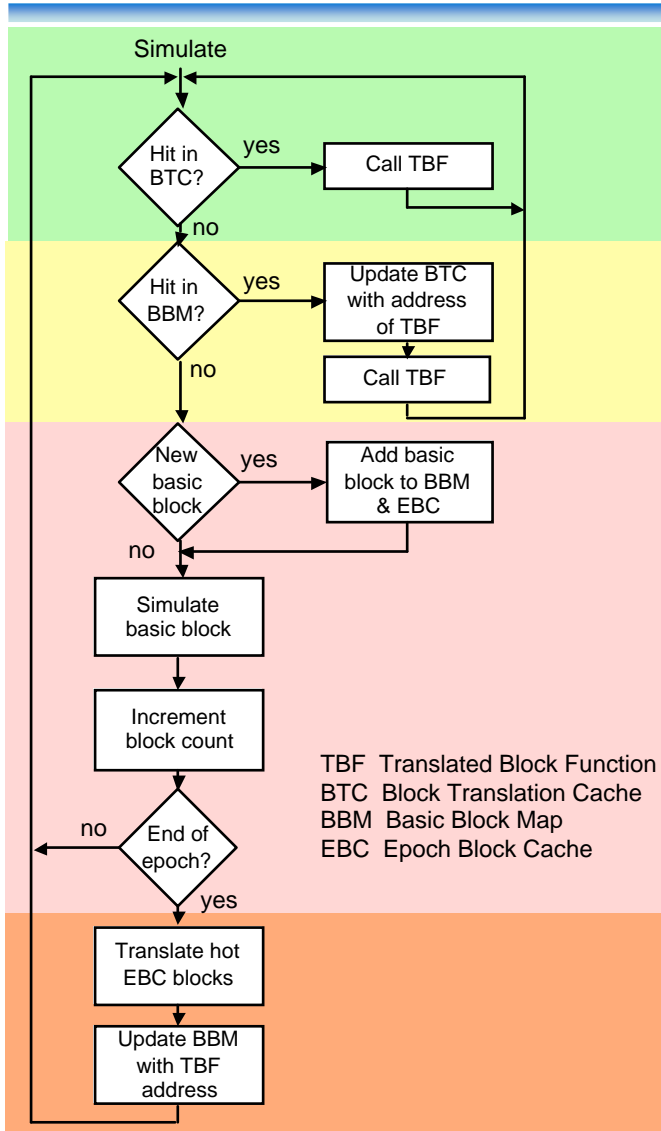
Research & development topics

- Retargeting the simulator
 - Currently ARC ISA for embedded systems
 - Modular replacement possible route
- Increased decoupling of DBT from the simulation engine
 - Perhaps using LLVM or similar
 - Trade-off between translation speed and peak performance
- Define an API for pluggable performance models
- Implement a broader set of performance and power models
- Interested in working with others in HiPEAC...



How the simulator works...at a high level

Simulation, Profiling and Translation



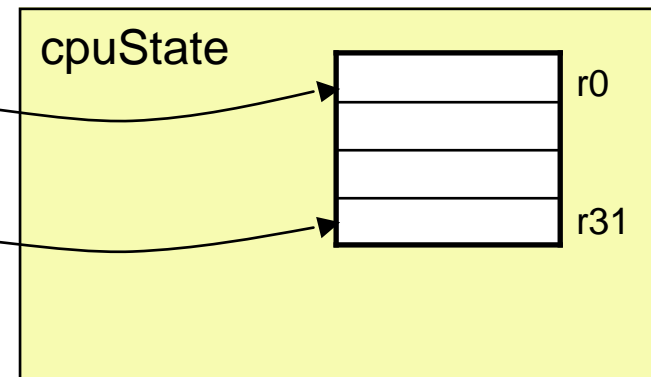
- Dual operating modes
 - Interpret, discover and profile
 - JIT translate and execute
 - Switches automatically between modes on basic block boundaries
- JIT mode
 - Find translated block in cache
 - Find translated block in map<>
 - Translate hot blocks to native (x86) code at each simulation epoch
 - Factor of 10x increase in speed
- Interpretive mode
 - Default if translation not present
 - Discover basic blocks dynamically
 - Profile basic blocks

Instruction Decoding and Caching

- Interpreter uses a 2-way set-associative Decode Cache
 - Each entry is a decoded instruction object
 - Indexed by PC
- Decoder creates instruction object
 - Designed to minimize effort during interpretation
 - Operands point to state elements according to format and operand value
 - Opcode selects semantic function
 - Retains enough detail to support JIT translation

add r0,r31,0xdeadbeef

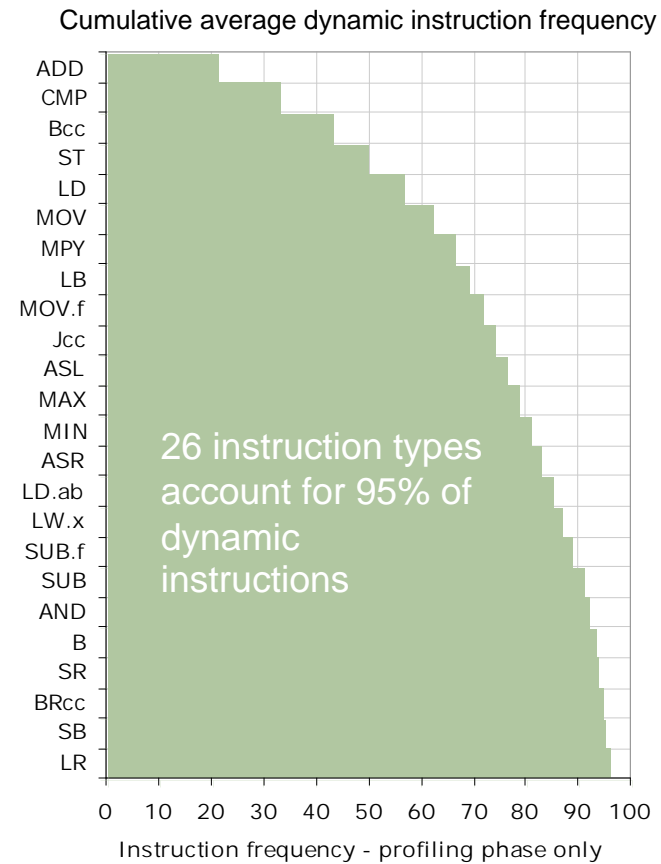
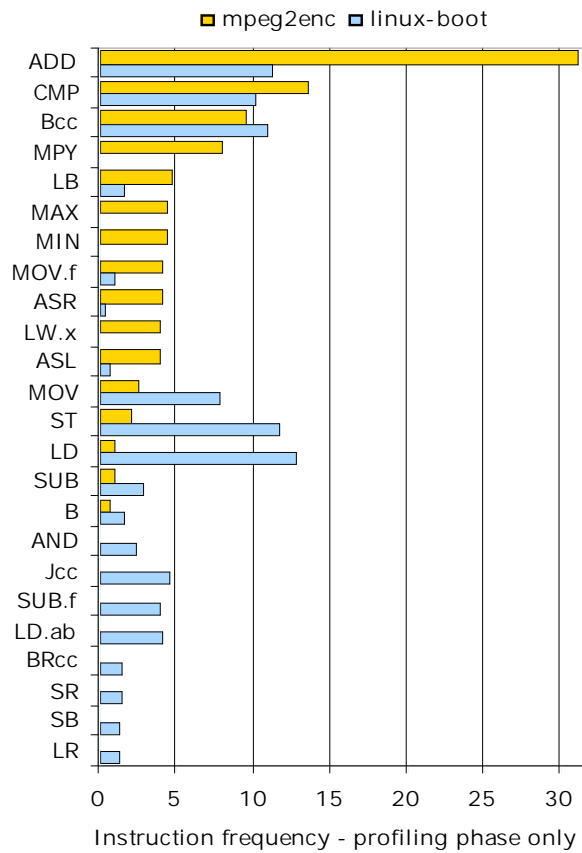
unsigned char	code	SIM_ADD
_uint32	*dst1	●
_uint32	*dst2	0
_uint32	*src1	●
_uint32	*src2	●
_uint32	limm	DEADBEEF



***dst1 = *src1 + *src2;**

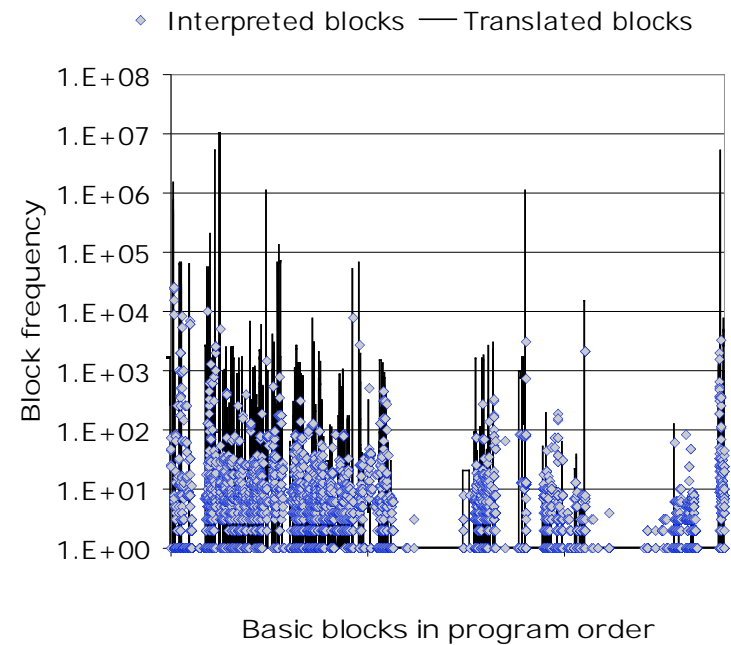
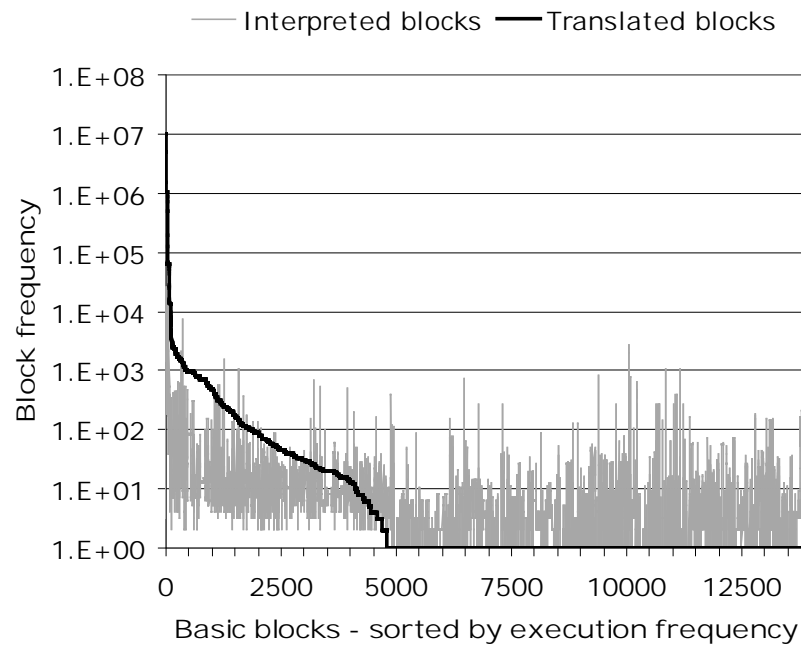
Dynamically-profiled Instruction Frequencies

- Simulator generates dynamic instruction profiles
- Informs simulator optimisation (interpretive and translated)



Identification of Hot Regions for JIT Translation

- Profile built for all interpreted blocks during each epoch
 - Epoch ends on completion of fixed number of interpreted blocks
 - Hot blocks identified for translation at end of epoch
- Linux kernel measurements:
 - 40.7% instructions translated
 - 99.8% instructions executed are JIT translated



Example Translation from Linux dcache_init()

- A simple loop computing $\log_2(r3)$
- Identified as 'hot' during booting of Linux kernel
- Translated during the 11th epoch of simulation

- Interesting features:
 - Has conditional branch
 - Branch has a delay slot
 - Includes a very simple instruction (adding 1 to a register)
 - Illustrates communication of ZNCV status from Compare to Branch

```
80007214 <dcache_init>:  
:  
800072ce: 2f 23 c2 00          232f00c2    lsr      r3,r3  
800072d2: 40 22 42 00          22400042    add     r2,r2,1  
800072d6: 4c 23 00 80          234c8000    cmp     r3,0  
800072da: f6 07 e2 ff          07f6ffe2    bnz.d   800072ce <dcache_init+0xba>  
800072de: 0a 24 80 00          240a0080    mov     r4,r2  
:
```

Translated code for example block

```
L_800072ce:
    pushl    %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
# lsr r3,r3 -----
    movl    20(%edx), %ecx
    shrl    %ecx
    movl    %ecx, 20(%edx)
# add r2,r2,1 -----
    incl    16(%edx)
# cmp r3,0 -----
    xorl    %eax, %eax
    cmpl    %eax, %ecx
    seto    268(%edx)
    setc    267(%edx)
    sets    266(%edx)
    setz    265(%edx)
# bne.d -10 -----
    movb    265(%edx), %al
    cmpb    $1, %al
    sbb    %ecx, %ecx
    andl    $-20, %ecx
    subl    $2147454238, %ecx
    movl    %ecx, 4468(%edx)
    movb    $1, 270(%edx)
# mov r4,r2 -----
    movl    16(%edx), %eax
    movl    %eax, 24(%edx)
    movb    $0, 270(%edx)
    movl    %ecx, (%edx)
    leave
    ret
```

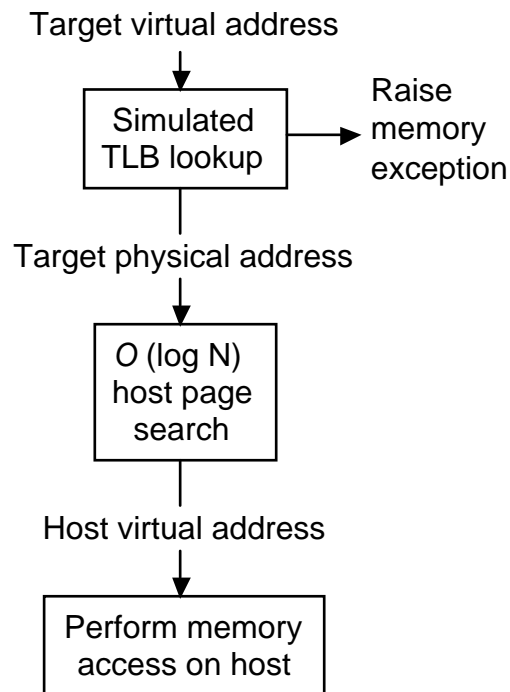
- 26 host instructions to implement the block
- 5 target instructions in the block
- ~5 host instructions per target instruction
- ADDing a constant is very efficient:
 add r2,r2,1 requires just 1 x86 instruction !
- Direct access to CPU state for registers and status bits, using **offset(%edx)**
- Note allocation of **%ecx** to **r3** for **lsr** and **cmp** instructions.
- Conditional branch does not itself contain any branch instructions, despite being coded with if-then-else.

Assembler comments were edited to fit the slide, but instructions are unchanged

Address Translation of Target References

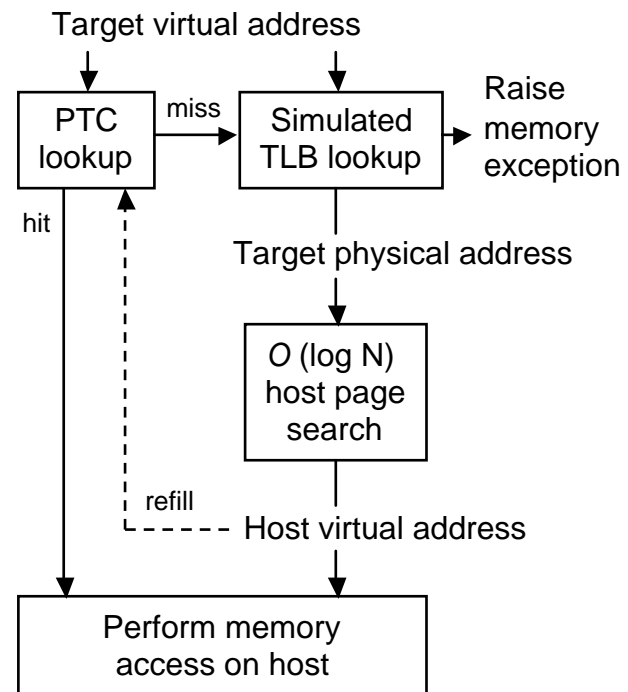
- Logical translation mechanism

- TLB lookup on every access
- Host page search on each access
- Slow and inefficient

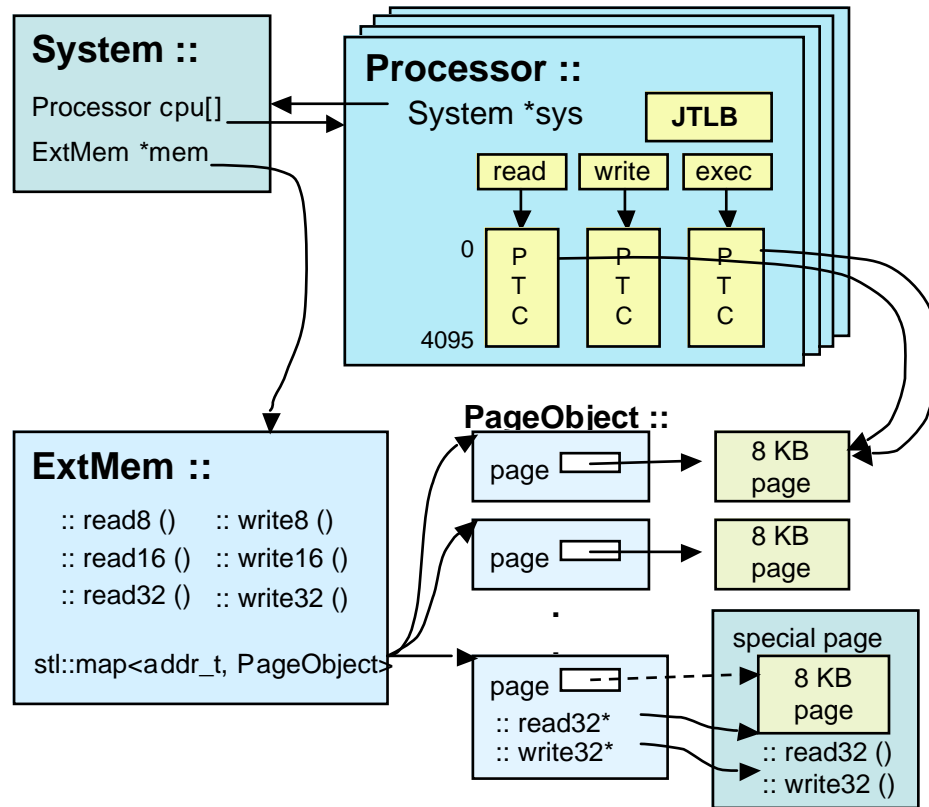


- Memoized translation mechanism

- Use a Page Translation Cache
- TLB + page search on cache miss
- Fast and efficient



Address Translation Structures



Address translation structures showing backdoor access via PTC

- Triplicate PTC: Read, Write, Exec
 - Easy coherence for SMC
 - Trap access violations
 - Refill models JTLB access
 - JTLB access raises TLB faults

- Detect special page access
 - Route accesses via special Read and Write methods

- Structures support SMP multi-core
 - Multiple Processor objects
 - Single ExtMem object
 - Atomic read-modify-write

Conclusions and Future Work

- Key features, compared with previous simulators:
 - We perform truly dynamic binary translation
 - We cope with self-modifying code and physical memory reuse (e.g. paging)
 - We retain a precise view of target machine state at all observation points
 - Simulator integrates with debuggers and ASIC verification frameworks
 - Provides graceful degradation when timing models are enabled
 - Reuse of persistent binary translations minimizes re-translation overhead
 - Includes execution-driven performance models
- Typical simulation speed of 450 native MIPS is ~4x FPGA speed
- Future extensions:
 - Multi-core simulation, on multi-core hosts
 - Power models
 - Pluggable modules for custom performance models
 - Evolution to a fully-retargetable simulator (ISA agnostic)

Multi-core simulation possibilities

- Functional simulation on multi-core host is trivial to implement
- Cycle-accuracy requires some synchronization at interaction points
- Simulator becomes a parallel application...

- Today, we achieve ~10 MIPS / host CPU in cycle-approximate mode

- We anticipate reasonably linear scaling to multi-core hosts
 - Degradation due to distributed-event synchronization
 - Probably still faster than FPGA approach