SimOS–PPC

Full System Simulation of PowerPC Architecture

Tom Keller
Austin Research Lab
IBM Research Division
tkeller@us.ibm.com

This Talk:(Inside IBM) http://simos.austin.ibm.com
(Outside IBM) http://www.ece.utexas.edu/projects/ece/lca/events
The SimOS-PPC Team

Full time

    Pat Bohrer
    Tom Keller
    Ann Marie Maynard
    Rick Simpson

Contractor (ex-AIX development)

    Bob Willcox

Temporary assignment (now completed)

    Brian Twichell
Full System Simulation
Background

Processor and system design has been focused around trace-based analysis, using traces (mostly) of problem-state programs.

- Traces that include operating system code are difficult to collect.
- Traces of multi-processors are extremely difficult to collect and to correlate.
- Traces tend to be old, because they’re difficult to collect.

From the traces, we generate statistics such as cache and TLB miss rates, code frequency-of-use, memory bandwidth requirements, and the like.

Tracing problem state code tells us nothing about —

- Execution paths through supervisor state code
  - Operating system services
  - Device drivers
- Cache traffic due to supervisor state code
- Cache traffic due to interrupt handlers
- Memory traffic due to I/O operations
For multiprocessors, trace-based analysis is even more problematic

- Problem state traces miss operating system code, especially code that deals with MP synchronization.
- Execution of code on one processor affects execution (and hence trace) on other processors (lock spinning, contents of shared caches, …).
- The usual trick, which is to ‘pretend’ that several copies of the same uniprocessor trace are running on the different processors, usually with different starting points, doesn’t model any of the MP interaction.
Full System Simulation

A way around the difficulties of traces is to use *execution based* simulation on a *full system simulator*.

A (mostly) unmodified operating system and interesting applications are run on the simulator.

- The simulator models —
  - Instruction set architecture
  - Caches
  - Memories
  - Busses
  - I/O devices
  - Multiple processors
- with enough precision to answer interesting questions, and
- rapidly enough to give answers in finite time.
Full System Simulation

Recent History

SimOS (Stanford University)
- MIPS R4000, R10000 + Irix
- Compaq Alpha + Unix
- [http://www.simos.stanford.edu](http://www.simos.stanford.edu)

SimICS (virtutech, spinoff from Swedish Institute of Computer Science)
- Sparc V8 + SunOS 5.x
- [http://www.simics.com](http://www.simics.com)
SimOS–PowerPC

Our project is a port of SimOS to AIX on PowerPC, with the addition of PowerPC processor simulators.

We model —

- PowerPC ISA (64-bit ‘Raven’)
- Caches, memory
- Selected I/O devices, sufficient to run a server workload:
  - Disk
  - Ethernet
  - Console

Each element can be modeled at varying levels of ‘faithfulness’, with an inverse relationship between accuracy and speed of model.

Example: Disk model

- Simple ‘instantaneous’, interrupt-free model used for AIX bring-up
- More complex model now in use models delays of actual disk access, interrupts at proper simulated time, models DMA transfers.
SimOS-PPC

Applications  32- or 64-bit

AIX 4.3.1

SimOS-PPC  64-bit PowerPC

Simple ISA simulator

Stable ISA simulator

AIX 4.x

PowerPC processor  32- or 64-bit
Adapting AIX to SimOS

AIX kernel

- Is unmodified. We use a copy we’ve built with –g so that all the debugging information is present.

RAMFS

- Contains drivers for our disk and console.
- Contains an ODM database with special ‘config rules’ to configure our drivers.

‘savebase’ information

- Built by us to describe the simulated machine configuration to the early stages of the boot process.
- Everything non-essential is removed, so that AIX doesn’t spend time trying to discover what’s on the bus.

All this is bundled into a boot image by a modified version of the bosboot command.
Adapting AIX to SimOS

Device drivers

- Interface with SimOS’ device simulators via a special PowerPC instruction
- Unassigned PowerPC opcode interpreted by SimOS as “simulator support” call (same trick as “diagnose” on VM/370)
- Interrupt-driven console, disk and ethernet drivers
- Console interface will remain simple, as it isn’t performance critical.
  (Simple doesn’t mean lack of function, though: it runs well enough for vi, smitty, and emacs.)

Files can be copied between the simulated AIX and the host environment:
```
simos-source /simos/src/tmp/ros/emacs.tar | tar xvf –
```
TCL interface to SimOS

TCL scripts are used to control the simulator

- Configuration (cache geometry, memory size, number of processors, …)
- Statistics collection
- Run-time control

Most TCL is in the form of *annotations*

- Run arbitrary TCL scripts at specified ‘points of interest’
- Specify where/when to run an annotation by:
  - Execution address (numeric, symbolic)
  - Load or store to specified address
  - Hardware event (device interrupt)
  - User-defined events
    - Creating a new process
    - Entering the idle loop
    - Dispatching a particular thread
    - …
TCL Annotations

Annotations are the basis of SimOS’s data collection.

Annotations have access to all the symbols of the program(s) being executed:

symbol load kernel unix

Through special TCL variables, annotations have access to the entire machine state:

REGISTER(regname)
MEMORY(virtual address)
PHYSICAL(real address)
CYCLES (current cycle count)
INSTRUCTIONS (current instr count on current CPU)

Examples (from IRIX):

Get the name of the current process:

symbol read kernel::((struct user*)$uarea)->u_comm

Count number of TLB faults via an annotation on the entry to vfault():

annotation set pc kernel::vfault:START {
    incr vFaultCount
}

IBM Austin Research Lab
Another IRIX example:

Tracking process fork, exec, and exit

annotation set osEvent procsstart {
    log “PFORK $CYCLES cpu=$CPU $PID($CPU)-$PROCESS($CPU)\n”
}

annotation set pc kernel::exece {
    set argv [symbol read “kernel:exec.c:(struct execa*)$a0)->argp”]
    log “PEXEC $CYCLES cpu=$CPU “

    # print out the whole command line
    set arg 1
    while {$arg != 0} {
        set arg [symbol read kernel::((int*)$argv)<0>]
        if {$arg != 0} {
            set arg [symbol read kernel::((char**)$argv)<0>]
            log “ $arg”
            log “\n”
        }
    }
    log “\n”
}

annotation set osEvent procexit {
    log “PEXIT $CYCLES cpu=$CPU $PID($CPU)-$PROCESS($CPU)\n”
}
Copy-on-write disks

The AIX disk image is copied from a real disk on a running AIX 4.3.1 system.
- The disk is not modified during SimOS execution.
- One disk image serves for multiple simulations, multiple users.
How Fast? (1)

SimOS-PPC’s ‘simple’ simulator running a cpu-intensive speech benchmark:

<table>
<thead>
<tr>
<th></th>
<th>32-bit 160 Mhz 604e</th>
<th>64-bit 125 Mhz Raven</th>
<th>64-bit 250 Mhz Blackbird</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated throughput, KIPS</td>
<td>156</td>
<td>240</td>
<td>340</td>
</tr>
<tr>
<td>Simulated KIPS Host MHz</td>
<td>.97</td>
<td>1.9</td>
<td>1.4</td>
</tr>
</tbody>
</table>
How Fast? (2)

SimOS-PPC “Simple” simulator running on an IBM 260 Mhz 64-bit host:

- CPU-intensive benchmark runs native at 1.6 cycles/instruction or 162 MIPS
- SimOS-PPC running on host emulates host at .34 MIPS, or 466 to 1
- SimOS-PPC running on host, modeling host’s L1 and L2 caches, executes at .095 MIPS, or 1713 to 1
- Simple simulator has not been tuned for performance.
- Block translator should improve the 466 to 1 case by a factor of 5.
Checkpointing

A checkpoint can be taken at any time
- After every \( n \) instructions
- When a specific point has been reached (via an annotation)
- By operator command at the simulated OS’ console

The checkpoint includes the entire state of the system:
- Registers and memory on all processors
- Cache contents
- Outstanding interrupt state
- Disk contents

Start-up from a checkpoint is immediate (about 2 seconds)

Repeatability for debugging and for ‘what if’ experiments with different configurations
What’s currently running

Simple CPU simulator

• One-at-a-time fetch/decode/execute loop
• Implements semantics of 64-bit Raven running as a uniprocessor, 2 or 4-way SMP
• General L1/L2 SMP caches modelled
• Idle loop is recognized; clock advanced to the next interesting event

Ethernet

• Simulated machines are seen as “real” to site, each with unique IP address
• Full Telnet and FTP. X11 is supported.

Simple model of disk (fixed delays)

Simple console

TCL interface for configuring the simulator
Lines of Code

- **SimOS Framework**
  - 600 Files
  - 95,000 lines

- **SGI MIPS**
  - 175 Files
  - 57,000 lines

- **GNU Environment**
- **Compaq Alpha**
  - 160 Files
  - 55,000 lines

- **64-bit AIX gdb**
  - 165 files changed
  - 10 files added
  - 14,000 lines added or changed

- **IBM PowerPC**
  - 180 Files
  - 47,000 lines
What’s next

1Q-99 Block CPU simulator for UP
- Translates and caches basic blocks
- Drops back to Simple simulator for ‘hard’ instructions (e.g., mtmsr)

2Q-99
- User program debugging with gdb. Running on a real machine, user attaches to a simulated process running in SimOS-PPC and debugs it.

3Q-99
- AIX system and user program profiling

4Q-99 & Beyond
- SimOS-PPC Support and IBM Proprietary Studies
# Customers & Status

<table>
<thead>
<tr>
<th>External to IBM Users</th>
<th>Principal Requirement</th>
<th>Additional Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of Texas</td>
<td>General architectural analyses tool</td>
<td>Integrate cycle accurate models</td>
</tr>
<tr>
<td>Carnegie Mellon</td>
<td>Front End for PowerPC MW simulator</td>
<td>Integrate Microprocessor Workbench</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IBM Customers</th>
<th>Principal Requirement</th>
<th>Additional Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix Performance Group</td>
<td>Software Tuning and Performance Debugging</td>
<td>Profilers and Instruction flow tracing</td>
</tr>
<tr>
<td>Future IBM Systems</td>
<td>High End SMP and Processor Design</td>
<td>Integrate cycle accurate models</td>
</tr>
<tr>
<td>OS Research</td>
<td>Platform for debugging new OS boot</td>
<td>None</td>
</tr>
</tbody>
</table>
The Block Simulator

Translates and caches sequences (blocks) of instructions

Almost all the simulated machine registers reside in actual registers
• r0, r2 – r15, r24 – r31, cr, xer, all FP regs
• Re-use of values in registers across separately-generated blocks

Block ended by
• Branch instruction
• Any instruction the block simulator can’t deal with

Generated code calls (via blrl) an assembly-language routine to handle
• Address translation for load/store
• Periodically checking for pending interrupts
• Complicated instructions (lmw/stmw, trap, …)
• Resolution of branch addresses
## Translated block example

### Original block:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10008154</td>
<td><code>rlwinm r6, r6, 2, 0xFFFFFFFFFC</code></td>
</tr>
<tr>
<td>10008158</td>
<td><code>lwzx r0, r7, r6</code></td>
</tr>
<tr>
<td>1000815C</td>
<td><code>cmpwi cr1, r0, 10</code></td>
</tr>
<tr>
<td>10008160</td>
<td><code>bge cr0, 0x10008180</code></td>
</tr>
</tbody>
</table>

### Translated block:

- Decr ctr, branch non-zero
  - `call support (time expired)`
  - `rlwinm r6, r6, 2, 0xFFFFFFFFFC`
- Add `r17, r7, r6`
  - `call support (translate for load)`
    - `lwzx r0, r0, r17`
    - `cmpwi cr1, r0, 10`
    - Add 4 to instruction count
    - `bge cr0, _______`
      - `call support (resolve branch fallthru)`
      - `call support (resolve branch target)`
Java Spec 1% MTRT

Percentage of Instructions

Interval is 1 Million Instructions
Java Spec 1% MTRT Memory Reads/Writes (Stacked)

Each Interval is 1M Instructions

Count

MEM_READ

MEM_WRITE
Spec Java 1% MTRT Disk Activity (Stacked!)

Each Interval is 1M Instructions

Count of Reads/Writes

DISK_WRITE

DISK_READ