Shade: A Fast Instruction-Set Simulator for Execution Profiling

Background: SPARC

- What are register windows? (motivation, function)

Register windows are hardware support for procedure calls. In a given function, the currently-running thread can access INPUT registers, LOCAL registers, and OUTPUT registers. INPUTs contain the arguments from a caller function; LOCAL are used for scratch computations; OUTPUT are used when passing parameters to a about-to-be-called function.

When a register window is saved (just before a call to a function), the hardware makes the OUTPUT registers into the INPUT registers for the called function, and gives it a new set of LOCAL and OUTPUT registers.

Given a fixed set of resources, sometimes a system will run out of register windows, in which case systems software must get involved and save all of them to memory (window overflow), and eventually restore them as the call stack unwinds.

- What are delay slots? Why are they useful?

Delay slots are the slot after a branch or jump which in older-style ISAs can be filled with an instruction to execute before the branch takes place. They result from the organization of pipelined implementations and are generally now considered odd to expose in an ISA.

- Why do some instructions get annulled?

Annullment makes branch delay slots more useful. If a branch is taken, an instruction in the delay slot will be executed; if not, it gets annulled. This makes delay slots easier to use in some loops and if-else statements.

- What does the CALL instruction do? How is the return address saved?

Call saves the return address in OUTPUT register o7, which becomes input register i7 after the register window is saved.

How is this different from a JMPL (jump and link)?

Jump and link is the general instruction that underlies call (and return); it allows you to specify which register to put the caller address into (as well as the target)

How is this different from a RET instruction?

RET is just another jump-and-link, this time to the address in i7 + 8 (just after the call and its branch delay slot)

Basics

- In contrast to Shade, how would a typical "classic" instruction-set simulator operate?

Classic simulators fetch, then decode, then execute. They are thus wildly slow.

- Describe, in contrast, the main operation of Shade; how does it work?

Shade generates the code to simulate the machine. It can thus spend a great deal less time checking things and more time in actual execution of the simulated code.

What is the "translation cache"? What is its structure and how is it used by Shade?

The TC is a critical data structure, as it holds the translated basic blocks.

What is the "TLB"? What is its structure and how is it used by Shade?

The TLB (poorly named) is a PC-to-VC hash table which allows Shade to lookup where in the TC a given translation exists.

Why is the notion of a "basic block" useful?

A basic block is a sequence of code starting at an entry point and ending in a jump or branch. Shade will normally translate one basic block at a time, and thus gain efficiency by jumping to this block and executing it without re-entering the main loop of the simulation.

- Figures 2a and 2b show an example translation (without tracing);
  How fast would Shade run if each instruction were translated like this?
  (how does Shade gain back some efficiency?)

It is slow because it has to fetch register values from memory; some of this loss is gained back through basic-block translation, which allows reuse of some of these values (i.e., keeping them in registers).

- How does Shade add tracing into translations?

Very awesomely. By embedding the exact tracing info into the TC itself, Shade only adds tracing overhead when you exactly need it. It also does so efficiently by smartly putting in trace points at the point where traced values are accessed, thus minimizing costs.

How is this done efficiently?

As above.

Chaining

- What is chaining?

Chaining allows Shade to jump from one translated block to the next without going through the main simulator loop.

- How does chaining work given the main loop in Figure 1?
- Follow the paths through the code for chaining when…
  … the successor was translated first
  … the predecessor was translated first

It is pretty cool but all you have to do is read the right paragraph in the paper to figure this part out.

Virtualization
- How are registers virtualized?
  What does Shade do in order to make this virtualization efficient?

Mostly discussed above. Registers kept in memory and
accessed as needed; can be kept in scratch registers in
a translated block to speed up things.

- How is memory virtualized?
  How is this different than the virtualization of memory we've discussed in other papers?

Much different and simpler. All in the Virtual Address Space of Shade simulator;
shared with simulator itself; offset by a fixed amount.

- What are condition codes?
  What makes them hard to virtualize?
  What is Figure 4 all about?

Condition codes are set implicitly by certain instructions;
they are useful to branch off of (conditionally).

They are hard to virtualize because sometimes you can't
access them directly (as in SPARC).

Figure 4 shows the assembly-level wizardry to extract CCs
on SPARC; it is just a cool small piece of code. It would
not exist if SPARC had a simple "load-this-CC" instruction.

- What does Shade do on system calls?

Just calls into the host OS. Easy when not running a cross shade.

Performance
- What do we learn from Figure 6?
- Figure 7?
- Figure 8?
- Anything else that interested you?

Didn't cover this in class.

Other
- Cross-shades: what and why? Useful in VMM setting?

Didn't cover.

Overall
- What is the coolest thing about Shade?
- What would it take to make Shade a VMM?
- Does Shade still exist?

Didn't really cover this either.