acmqueue

Enhanced Debugging with Traces

An essential technique used in emulator development is a useful addition to any programmer's toolbox.

Peter Phillips

Creating an emulator to run old programs is a difficult task. You need a thorough understanding of the target hardware and the correct functioning of the original programs that the emulator is to execute. In addition to being functionally correct, the emulator must hit a performance target of running the programs at their original realtime speed. Reaching these goals inevitably requires a considerable amount of debugging. The bugs—which for players of old arcade games are a vital part of re-creating the original experience—are often subtle errors in the emulator itself but could also be a misunderstanding of the target hardware or an actual known bug in the original program. (It is also possible the binary data for the original program has become subtly corrupted or is not the version expected.) Solving these problems requires some unusual debugging techniques.

The debugger of the host system can be used for the target (emulated) system by setting a breakpoint in the CPU do-instruction loop and examining the variables that hold the contents of the emulated CPU registers and main memory. This works for small and easily localized problems such as failures in the self-test or initialization code but is not of much use for problems that happen at a later stage in the execution. At that point the organization of the program is not apparent. No debug symbol or source code is available to allow a high-level approach to the problem. More powerful techniques are required and are built into the emulator itself.

Obviously there is enormous variability in the kinds of bugs that may present themselves. Ignoring major execution path failures (e.g., infinite loop, the program counter ending up in nonprogram memory), a typical failure would be noticed in the video output of the target, shown in figure 1. In this example a sprite's location is definitely incorrect. The top sprite is offset from being in the expected column and is at 13,11 instead of 24,11. We know the aliens always attack in columns, so something is wrong with the emulator. A pair of values in the emulator's memory state determines the sprite's position. This address would, in fact, be the location of a video hardware register that directly determines the displayed position. You can use the platform debugger to confirm the register values are, in fact, 13,11. Unfortunately, even if you know the values are wrong, finding the actual error is not easy.

The incorrect register values are the end of a chain of earlier calculations (see sidebar, "Screen Image Chain"). Finding the error involves working backwards through the causal sequence to find where the miscalculation occurred. The debugger, however, shows us only the current state of the machine. If it is possible at all, reconstructing the event chain will involve a lot of detective work.

TRACES TO THE RESCUE

There may be an easy shortcut to solving the problem. If you are trying to create a faster emulator from an existing one that is known to work, then you can use the correct emulator to debug the incorrect one. The correct emulator can be modified to create a trace of input and CPU state, and

the target emulator can read this trace file (see sidebar, “An Example 6502 Trace”). The input values are used instead of real input (joystick positions). After each `do-instruction` call, the current machine state is matched with the trace, and execution is halted on divergence. At this point you can use the host debugger to examine the problem in detail and find the resolution.

What if, as is more likely the case, you don’t have a reference trace? You still know the ultimate source of the error is when the incorrect X value 13 is placed into memory location `0xc000`. You can search the trace file to find cases when this value is written (e.g., search the `BUS` column for `C000w0d`). At this point you will see the incorrect value coming from a register, and looking just a few trace lines back, you will identify the address of the incorrect value in the `shadow[]` memory (see “Screen Image Chain” sidebar to show the path being followed).

At this point you can apply the same steps with the (now) known address of the shadow memory to find the point in the trace where the *sprite* struct is modified. As you keep tracing backwards, you will find either more memory locations to trace or sections of code that can be hand-verified to find the offending emulation error. It certainly isn’t as easy as using a reference, but there is a good chance of finding the error without constantly rerunning the emulator.

FIGURE 1

A Sprite Misplaced

USING TRACES

An emulator using traces can hunt down problems in a machine description, but traces can also be useful when the problem exists in the code itself and not the machine description. Using such a trace can make finding memory corruption problems much easier. Consider a particularly nasty kind of corruption: the corrupted data structure is owned by the runtime memory allocator implementation. For example, the program crashes inside of a call to `malloc()` as it traverses the free list. The actual crash is apparent in the debugger but the ultimate cause is not.

Suppose that the direct problem is that a particular pointer value at location `0x8004074` in the

Screen Image Chain

In bits of partial C code, here is how a sprite position is determined, starting from the actual hardware value and working backwards. The C code is reverse engineered, as the emulator would execute machine instructions directly.

1. The image position of sprites is determined by hardware registers starting at `0xc000` (each register has a size of two bytes):

```
Sprite #0 0xc000 Xpos Ypos Shape Palette
```

```
Sprite #1 0xc008 Xpos Ypos Shape Palette
```

```
...
```

In this case, `0xc000` contains `0x000d 0x00b` (corresponding to 13,11 on screen).

2. The registers are set at the vertical blank (60 times per second) frame by copying from an internal array.

```
struct sprite { unsigned short x, y, shape, palette; } shadow[16];
```

```
...
```

```
void vblank_intr() { memcpy( (void*)0xc000, &shadow, sizeof(shadow));
```

3. The shadow values are updated in an object update routine.

```
void Object::update_sprite() {
    shadow[this->sprite_index].x = this->x + this->x_center;
    shadow[this->sprite_index].y = this->y + this->y_center;
}
```

4. The object changes the position based on speed.

```
void Object::move() {
    this->x += this->delta_x;
    if (this->x > 128) { this->x -= this->delta_x; }
    ...
}
```

The fourth and final level is the first point of interest. Perhaps something is wrong with the emulator's implementation of the compare instruction.

allocator's linked list has been modified. Instead of a plausible value, it contains 0x1—an invalid and unaligned memory reference. The standard debugger cannot be of much more help at this point, as it can reveal the invalid value but not when it became invalid. (You could use a break-on-write, but that won't work if the corrupted address changes between runs.)

Here is where the trace solves the problem. Search the trace backwards for the last write to the memory location 0x8004074. You may find an entry such as:

0x80004074: write 0x1 PC=0x1072ab0 ...

Translating the given program counter (0x1072ab0) to a location in the source code will immediately reveal a likely cause of the corruption. For example, it may now be clear that the array bounds were not correctly respected. An attempt to store a value of 1 in the array resulted in the memory corruption. Of course, as in the sprite example, the actual source of the corruption may need to be traced further back.

TRACING FOR ALL PROGRAMS?

Tracing is useful for debugging ordinary problems, but how to generate these traces isn't immediately apparent. Unlike the emulator, you do not control the (hardware) implementation of the microprocessor running your software.

One may object that such verbose logging is not particularly feasible for “real” programs. Although hard-drive storage continues to decrease in cost, I/O bandwidth does have limits. If storage is truly limited, there are a few other approaches. For example, the traces between snapshots could

An Example 6502 Trace

The 8-bit 6502 microprocessor had only a few registers: A, X, Y, S, PC, and P. An actual system might also have a simple 8-bit input port for joystick positions. The trace file would be (hexadecimal) values of the registers and I/O port along with the current (symbolic) instruction:

```
PC  A  X  Y  S  P  IO  BUS  ASM
F010 00 01 02 F7 D1 01 F011r7A lda #$7A
F012 7A 01 02 F7 31 01 ...
```

In this example, the instruction at location F010 loads the value 7A into the A register, and the next line of the trace reflects the change. The high bit of the status register (named P on the 6502) is cleared as the value in A is not negative. A BUS column is included to show the reading or writing of data values onto the system bus.

There are a lot of possible ways to represent traces. For long traces, it may be more efficient simply to store entire machines states with a counter describing the number of instructions to execute before a new machine state is reached or an I/O value changes. The detailed trace can be reconstructed (as necessary) by running the emulator for N steps on a given machine state.

be stored only in RAM and not written to the log file. Traces could be re-created as necessary (by rerunning the target from a given state). Alternatively, only the most recent trace could be stored in main memory, providing quick access to scanning the trace if necessary.

The complete immutability of our CPU implementation is an elaborate illusion in many cases. All Java (and .NET) programs are actually running on virtual CPUs. Modifying these virtual machines to record trace information and state snapshots is conceivable—no hardware would have to be changed. It is merely a matter of convincing the owners of the virtual machine implementations to make the necessary changes.

Even native executables are a step removed from the real hardware. All modern operating systems enforce a separation between user and kernel modes of execution. A process is, in fact, a virtual entity. The capability already exists to snapshot a process's execution state (Unix's venerable core dump). With some additional operating-system support, it is quite feasible to take these snapshot states and restore them as real processes that can continue their execution.

Even the entire machine being used may be virtual. Products such as VMware and Parallels routinely operate with machine-state snapshots that include the entire state of user mode, kernel mode, device drivers, and even hard-disk drives.

HIGHER-LEVEL TRACES

Tracing memory accesses is helpful for programs written in languages such as assembler or C. Memory and CPU transfers correspond easily with actual source code, but it is more common to be working with languages a step removed from the machine, where the C code is an interpreter of the actual language in use. In this case the low-level action of the code does not easily map to recognizable actions in the interpreted language.

The use of I/O is also more complicated in modern systems. Programs do not read and write directly to hardware I/O locations. Instead, device interaction is mediated through the operating system.

Tracing can be applied to these higher-level programs in an obvious fashion: change the trace to record higher-level events. Exactly which events to capture would depend on the kind of program. A GUI program may need to capture mouse, keyboard, and window events. A program that manipulates files would capture open/close and read/write operations. Code built on top of a database might log SQL statements and results.

A good trace differs from a simple log of events. The trace must provide enough information that the correctness of execution can be verified using only the trace. It should be possible to construct a *reference implementation* that can read the trace and automatically verify that the correct decisions are made. Experience with emulators suggests that you might first code the reference implementation and use it to verify the production code. (You may avoid the premature optimization trap and discover that the simplified reference implementation is a satisfactory solution.) Writing a reference might seem to involve as much effort as the production version, but there are always requirements that do not change the functional output. For example, the production code may require a lookup table to be persistent, whereas the reference can use a simpler in-memory hash table.

CONCLUSION

Adding snapshots, tracing, and playback to existing debugging environments would significantly reduce the time required to find and correct stubborn bugs. Low-level code is seeing some progress in this area; for some platforms, gdb has recently been given the ability to reverse execution. Since CPU operations are not reversible, this means there are now ways of capturing trace information for compiled programs. If the addition of saving and reloading snapshots were added, gdb could become a traceable debugger.

Detailed CPU state traces are extremely helpful in optimizing and debugging emulators, but the technique can be applied to ordinary programs as well. The method may be applied almost directly if a reference implementation is available for comparison. If this is not the case, traces are still useful for debugging nonlocal problems. The extra work of adding tracing facilities to your program will be rewarded in reduced debugging time.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PETER PHILLIPS received a bachelor's degree in computer science from the University of British Columbia. He has 15 years of experience in software development and has spent the past 10 years of that working on console games.

© 2010 ACM 1542-7730/10/0300 \$10.00