Innovative Technology for Computer Professionals

Computer

May 2006

The Device Software Engineer's Best Friend

Michael Lindahl, Green Hills Software

Hardware trace can make developers' lives a little less stressful.

As embedded devices grow larger and more complex, the ability to debug the software that drives them will limit how much software can be successfully integrated to meet release deadlines. Further, as more device software consists of third-party or even open source programs, the challenge of integrating and releasing embedded devices on time becomes more daunting.

Fortunately for device software engineers facing this difficult environment, newly designed tools and technologies make this process easier. One technology becoming more pervasive, *hardware trace*, can significantly improve the integration and debugging process. Hardware trace offers a complete history of the instructions a microprocessor executes and provides unique visibility into the workings of device software.

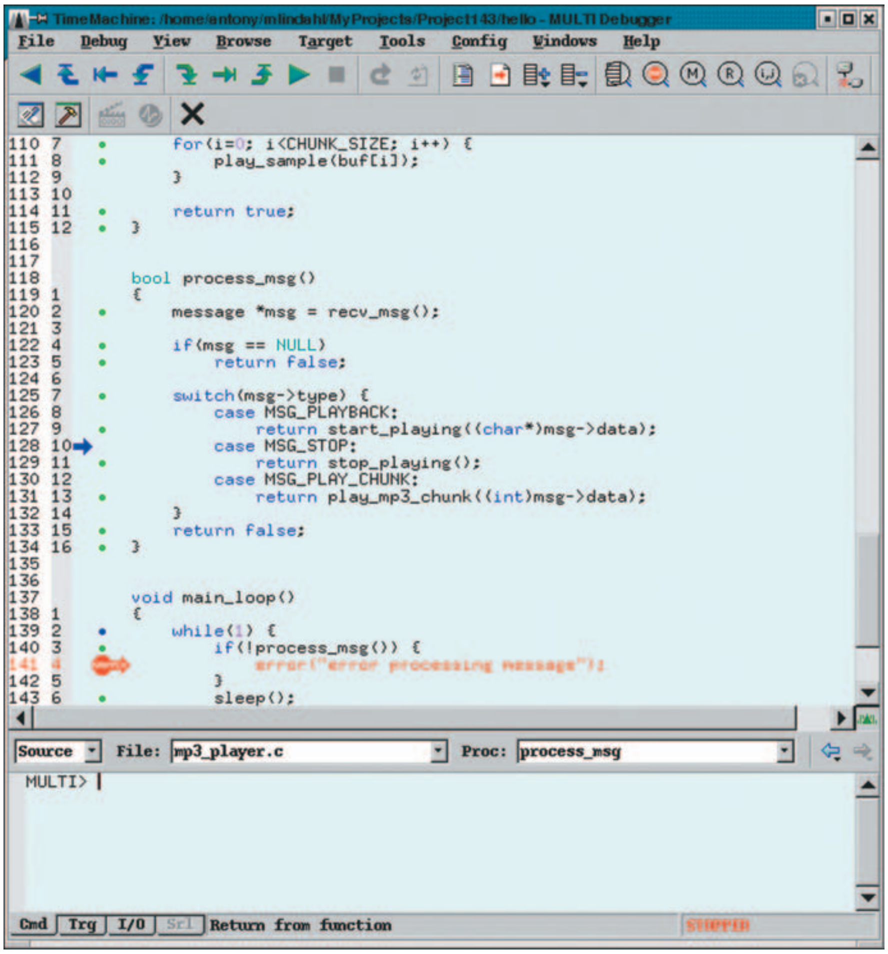
Traditionally a low-level hardware and firmware tool, hardware trace required that users pore over hundreds or thousands of lines of assembly code to determine a bug's source. However, a new generation of trace tools now presents this mass of information in terms software engineers can easily understand. Software engineers can use these tools to debug more efficiently by stepping and running programs backward in time using the same familiar interface as a standard software debugger. Some trace tools also provide advanced interfaces that help engineers quickly comprehend how their software works.

DEBUGGING NIGHTMARE

The most difficult software bugs occur only intermittently when testers try to reproduce them. These bugs can often be the source of lengthy schedule delays because they are unpredictable and might never be found. They are also the most likely to make it through final release testing, causing frustration for customers and expensive support problems for the developers' organization.

Using traditional tools makes the job of finding these bugs easier, but tracking them down and fixing them often remains extremely difficult. For example, if a program hits an error condition in software, setting a breakpoint on that line of code to catch the program after the error occurs can be done easily. Unfortunately, this does not let testers determine how the program arrived at the error condition. They still do not know what events led up to this error. It could be a memory corruption or some other type of intermittent problem. Or the hardware might have malfunctioned, feeding the software bogus data that was not properly handled. While testers can track down these types of bugs with traditional debugging tools, doing so is often frustrating and expensive.

If employed to capture detailed information about the system instead of simply applying traditional debugging techniques, hardware trace can locate the source of most errors relatively quickly. For example, if testers collect a trace leading up to an error condition, they can then move backward through the execution of their


TimeMachine: /home/antony/mlindahl/MyProjects/Project143/hello - MULTI Debugger
File Debug View Browse Target Tools Config Windows Help
Source
File: mp3_player.c
Proc: process_msg
Cmd Trg I/O Srl
STOPPED

```
        for(i=0; i<CHUNK_SIZE; i++) {
            play_sample(buf[i]);
        }

        return true;
    }


    bool process_msg()
    {
        message *msg = recv_msg();

        if(msg == NULL)
            return false;

        switch(msg->type) {
            case MSG_PLAYBACK:
                return start_playing((char*)msg->data);
            case MSG_STOP:
                return stop_playing();
            case MSG_PLAY_CHUNK:
                return play_mp3_chunk((int)msg->data);
        }
        return false;
    }


    void main_loop()
    {
        while(1) {
            if(!process_msg()) {
                error("error processing message");
            }
            sleep();
```

```
MULTI>
```

```
Return from function
```

Figure 1. Tracking an intermittent bug. A debugger hit an error when processing a simple MP3 playback message. When such a bug occurs only intermittently, it can be difficult to catch the conditions leading to the error. If they have hardware trace data leading up to the error, however, testers can step backward through the code to determine the error's cause.

software to locate the error's source. Tools like our TimeMachine Debugger integrate the trace data directly into the debugger, making it easier to find most bugs and perhaps even to find the most difficult, intermittent bugs using a familiar software debugger interface.

In an MP3 player, for example, testers could run into an intermittent bug during playback or a bug in the USB driver that causes the USB interface to freeze. These types of bugs can be solved relatively quickly using trace tools because testers can simply work their way backward from the bug's symptoms to the bug's root cause.

If an error appears because a variable has an inconsistent value, testers can work backward to where that variable was last written. This can easily be determined from trace data by setting a data watchpoint on the variable and running backward. When the debugger stops, testers can check the value of the variables that determine the incorrect value. Iteratively following this process will lead to the source of the bug, making it trivial to track down an otherwise difficult glitch.

Figure 1 shows an example in which a debugger hit an error when processing a simple MP3 playback message. If this bug occurs intermittently, it might be difficult to catch the conditions that led to the error. However, if testers have hardware trace data leading up to the error, they can simply step backward through the code to determine why it hit this error. After stepping back a few times, they arrive at the source of their problem, which was caused in this case by trying to read past the end of the MP3 file being played.

The file has 10 chunks in it, and the software is trying to read too much data from the file. While simple, if this bug only occurs in certain files or in limited and difficult to reproduce circumstances, hardware trace provides a valuable tool for finding and fixing the error.

A PERFORMANCE MESS

Intermittent performance problems can balloon into a huge mess. If, after fixing all known bugs in a system, testers discover that the software performs great most of the time, but some parts of the code seem to run too slowly under certain circumstances, tracking down the problem's source becomes difficult. Additionally, if developers are simply integrating third-party software, it becomes even harder to find the error's source because they must first spend a significant amount of time learning how the code works in detail.

With traditional debugging tools, they might set some breakpoints in a debugger or modify the code to attempt to catch its slow execution. However, once they determine that the code ran too slowly, they might have already passed the point at which they can easily discover the problem's cause. In addition, if testers use a software debugger or other tool that modifies the system's execution, the software might behave differently than when it runs at full speed.

With hardware trace, however, a system can run at full speed while testers nonintrusively collect information about what the software is doing. They can then analyze this information to determine why it failed to run optimally.

One example of a performance problem in which hardware trace is especially useful occurs when the amount of time a function takes to execute depends on its parameters. This is a common situation, but one that traditional profilers do not generally make easy to track down.

For example, say that the testers have finished the first version of the software for an MP3 player, but users complain that the device sometimes freezes momentarily when they add a new song. Because this does not happen every time, the bug is difficult to track down. However, if the testers can collect trace data for their system and then stop trace collection when the performance bug occurs, they can easily determine the source of this bug.

Using the PathAnalyzer tool, which shows a graphical view of the call stack over time, testers can see the results of tracing the application in Figure 2. Notice that there are several calls to add_song(), some of which take a long time and others that take virtually none. Zooming in on the worst case reveals that the longest call to add_song() takes roughly 33,000 cycles, compared to the average duration of 82 cycles. Trace analysis tools calculated this data to find each call to add_song(), then displayed the results.

This application uses a doubling array to hold its songs, so while on the average it takes 82 cycles, occasionally the extended time needed to add a song to the array leads to the delays reported by the application's initial customers. While this problem certainly could have been found with traditional tools, the trace tools make it easy to track and find once the data has been captured and analyzed.

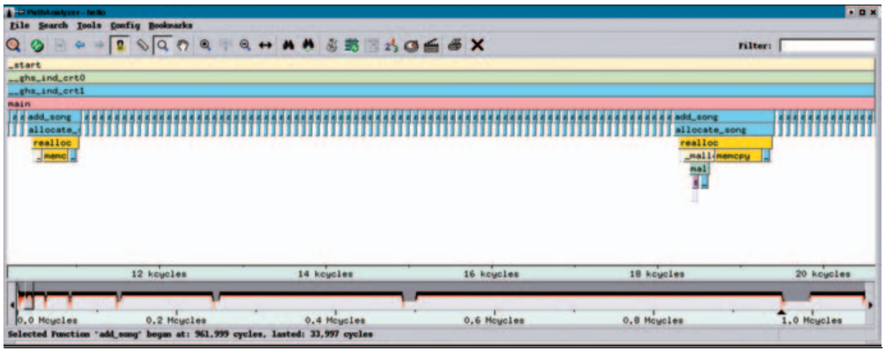
In addition, tools like the PathAnalyzer help testers track, identify, and fix performance problems where one example of a function takes a long time. Once they identify the problem, testers can modify their data structure relatively easily to prevent any single insertion from taking significantly longer than the others, which is the case with a doubling array.

HARDWARE TRACE

These powerful tools use the information that hardware trace ports provide. Several microprocessors provide such a port, which is generally a dedicated debug channel that includes information about each instruction the processor executes and, sometimes, even the data that the software running on the processor reads and writes.

Most hardware trace ports output information about the instructions they execute. Referred to as *instruction trace*, this process enables reconstruction of the flow of instructions the processor executes. In addition, some hardware trace ports output information about the data the processor reads and writes. This process, called *data trace*, traces the flow of data through the system.

To collect trace data, devices called *trace probes* gather data from a microprocessor's trace port and upload it to a host PC that translates this information into human-readable form. Once it has processed the data, the system feeds it into trace analysis tools that testers use to leverage the trace data for debugging and optimizing a system.



Selected Function 'add_song' began at: 961,999 cycles, lasted: 33,997 cycles

Figure 2. Debugging intermittent system freezes. Several calls to add_song() show that the worst-case add times take roughly 33,000 cycles, compared to the average duration of 82 cycles. Trace analysis tools processed this data to find each call to add_song(), which isolated for debugging the small percentage of songs that took much longer to add.

USING HARDWARE TRACE

Developers who want to use hardware trace data on their next project should choose a microprocessor that has a hardware trace port. The most widely available trace port, the ARM ETM, can be found on a wide variety of ARM microprocessors, including ARM 7, 9, and 11 systems. The ARM ETM provides full instruction and data trace, which testers can use to track down difficult bugs and performance problems.

In addition to the ARM systems, several other microprocessors provide either instruction trace or instruction and data trace functions, including the IBM PowerPC 405 and 440, and the PowerPC 55xx family of processors from Freescale. If the system lacks a hardware trace port, several options can simulate it, including instruction-set and system simulators and instrumentation solutions. Although none of these systems offer all the benefits of a nonintrusive hardware trace port, they at least let testers take advantage of trace analysis tools for several problems without having to migrate the system to a trace-enabled microprocessor.

As embedded devices become increasingly complex, time and quality pressures have become one of the greatest challenges facing device software organizations. Although no solution will banish all of a software engineer's problems, hardware trace offers a valuable tool that can help solve many of them. Whether the challenges consist of intermittent bugs or serious performance problems, hardware trace offers unique visibility into the workings of software that lets developers deliver better products in less time and with less risk.

By applying hardware trace to their projects, such as the MP3 player examples presented above, software engineers could find the challenge of integrating unfamiliar components into a complete device software solution a little less daunting. ■

Michael Lindahl is the engineering manager for Green Hills Software's TimeMachine debugger. Contact him at mlindahl@ghs.com.

Editor: Wayne Wolf, Dept. of Electrical Engineering, Princeton University, Princeton NJ; wolf@princeton.edu