A REAL-TIME MICROPROCESSOR DEBUGGING TECHNIQUE

Charles R. Hill

Computer Systems Research
Philips Laboratories
Briarcliff Manor, N.Y. 10510

ABSTRACT

This note describes RED, a remotely executed debugger capable of generating a real-time source level trace history of a high level language program executing on a microprocessor. The trace history consists of a display of the source statements of each basic block executed, annotated by the time at which execution of that block began. Basic blocks are traced rather than statements to reduce sampling bandwidth requirements while still retaining the ability to record the essential logical flow of programs. RED is intended to assist in debugging stand-alone high level language process control programs with real-time constraints.

We outline two possible implementation schemes for generating the real-time trace history. In both, a "debugging co-processor" collects in a history buffer the values of the program counter (PC) and the corresponding value of a clock as each basic block begins execution. The debugger, which runs on the processor hosting the compiler and has access to the co-processor over a fast link, reconstructs a source level trace from the PC-time pairs in the history buffer. In one scheme, the language compiler emits an extra instruction at the beginning of each basic block in the program to output the value of the program counter to a parallel port connected to the debug processor. The second method makes use of an extended target memory space to provide tag bits denoting basic blocks. When an instruction is fetched, the debug processor detects the presence of the tag bits and buffers up the value of the corresponding program counter and time. The first method is simpler to implement, requiring only conventional, usually straightforward hardware additions to the target, but requires the execution overhead of the extra instructions. In both cases the debugger itself runs on the host processor and has access to tables generated during compile time of the source program.

1.0 Introduction

Programs that have real-time constraints are among the most difficult to test and debug because there is an extra criterion for correctness that is not readily apparent from the program text - that of time. Since such programs depend on asynchronous external events, it may be impossible to compute at compile time the actual time one block of code executes relative to another. Simulation can be used to do some timing analysis and debugging. However, the detection of many bugs requires the presense of the actual target hardware. A means of tracing the execution of the program on the target and providing the actual execution times (without altering them appreciably during the measuring process) would be valuable to the real-time programmer for detecting both timing errors in the code and invalid assumptions about the timing of external events. The trace by itself would be useful for program testing.

Traditionally, time critical parts of microprocessor process control programs have been coded in assembly language to permit the programmer complete control over which machine instructions are executed and thus give him a more precise notion of how long the code takes to execute. It also permits him to optimize the code by hand to reduce its running time. However, as optimizing compiler technology improves, and the power of microprocessors increases with corresponding decreases in cost, it becomes more feasible and economical to use high level languages to code all parts of a system. Most microprocessor development systems still orient their real-time measurement toward machine language programs. In this note, we propose to extend real-time measurement to high level language programs in a form meaningful to the high-level language programmer.

2.0 Real-Time Debugger

RED is designed to be used with a high level statement oriented language for developing stand-alone process control software running on a

microprocessor. The debugger will have the traditional features such as inserting breakpoints and examining and depositing values of variables. In addition, it contains a real-time trace facility enabling the user to determine from a source level display which basic blocks were executed and at what time, relative to a real-time clock. Since that feature is what is new and unique about RED, in this note we concentrate on describing the real-time trace.

The software development is intended to take place on a a fairly powerful host, running UNIX.[1] The target processors are typically single-board computers, and the programs are down loaded for actual execution. The debugger is "remote" in the sense that the greater part of it runs on the host computer. This part includes the more complex debugger functions, such as the user interface, command parser, and command executor, which require access to compiler generated files. A small debugging kernel, which handles communication with the host and implements low level functions such as examining specific memory locations and handling breakpoints, resides on the target. RED also uses an intermediate processor between the host and target which we call a "debugging co-processor". This processor does the sampling for the trace. It also contains a high speed interface to the host processor such as an Ethernet or parallel link. This link does not need to operate in real-time, but must have sufficient bandwidth to send the contents of the trace buffer and the debugging command transactions to the host without undue delay. We believe a real-time debugging co-processor such as this is practical because of dropping memory costs and the existence of low-cost reliable high-speed links like the Ethernet. The latter makes it possible to host the debugger on a variety of machines, including large minis and mainframes.

2.1 Characterizing Control Flow Using Basic Blocks

A basic block is a sequence of straight line statements, having only one entrance and one exit; there are no transfers of control into or out of the sequence except by the one entrance and one exit. A procedure call or decision statement thus terminates a basic block. Since every statement within a basic block in a sequential program is executed the same number of times, the control flow is completely characterized by the sequence of basic blocks executed. Counting the executions of each basic block has been used in a number of compilers to efficiently implement post-execution summaries [Satt][Kieb]. On most machines, this can be accomplished with only one extra instruction per basic block.

Tracing the execution of basic blocks rather than statements has two main advantages for real-time tracing. First, bandwidth requirements are minimized, since sampling need not occur as often, and thus memory for buffering is reduced. Secondly, basic blocks are less likely to be re-ordered by code improvement techniques than individual statements. We claim that tracing basic blocks is a good compromise between tracing every statement, which is probably not necessary for timings and might be a burden on the sampling hardware, and only tracing procedure calls, which is probably too coarse a granularity.

Note that for the class of programs for which the trace facility is intended, i.e., stand-alone process control programs, all interrupts are explicitly handled by the high level language program. Thus interrupts in the middle of basic blocks will be visible by the trace of the interrupt handler routines, although the precise statement where the interrupt occurred will not be shown.

2.2 Trace Implementation

The trace consists of a display of each basic block that has executed over a specific interval, annotated by the time execution of that block began. This interval can be specified to be relative to the beginning of program execution, or from a breakpoint. The interval terminates when a breakpoint is reached, or the program terminates. The successful use of this technique depends on the presence of naturally quiescent points in the program at which the program can pause and not have to run in real-time.

The trace display itself is not generated in real-time, but is available when the program reaches the terminating breakpoint. The trace output consists of a pretty-printed display of the program and a cursor that marks the basic block currently being considered. The pretty-printing is performed so that distinct basic blocks always begin on a new line. A set of editor-like commands are provided to move the cursor, and hence change the current block. There are commands for displaying the next (or last) block executed, the next (last) textually contiguous block, the next (last) procedure, and to exit a compound statement. Each distinct visit of a basic block is annotated by the value of the time it was executed.

The debugger uses a data base generated by the compiler when the program was compiled, and the history recorded during execution. The compiler generated debug files, in addition to containing the usual information necessary for a symbolic debugger, contain the pretty-printed source text for each basic block, and a table of pointers into that text keyed on the address of the beginning of each basic block. (If the program was relocated by a linker, the linker would need to relocate the entries of this table). The only information that needs to be sampled at run-time is the starting address of each basic block and the time.

Two methods are described for performing the real-time sampling. One makes use of a parallel port attached to the target processor, driven by instructions generated by the compiler at the beginning of each basic block; this is called the "software probe" method. The second ("hardware probe") method relies on an expanded target memory space that contains tag bits denoting which

[1] UNIX is a trademark of Bell Laboratories.

instructions begin basic blocks. The addresses of basic blocks that have been executed can then be directly extracted by hardware.

2.3 Software Probe Method

The compiler inserts extra instructions in the beginning of each basic block as for the execution summary described above. However, instead of an instruction to increment a counter, it generates instructions to output the value of the program counter (PC) (or a basic block number) to a parallel port. (This would only require one instruction on processors allowing memory-mapped I/O). The parallel port is connected to the debugging processor D which buffers each value of PC received from the target (T) along with the current value of a high resolution clock. When the target program is in a quiescent state (either not yet initiated or halted at a breakpoint), a small kernel is in control of T. This kernel is capable of sending and receiving messages sent along the parallel line, and thus responding to simple commands sent by the debugger. These would be commands to send back or alter the values of memory locations and insert breakpoints (calls to the kernel) on T. When the user wishes to initiate a timing operation, he sets a breakpoint, and issues a debug command to trace. The debugger directs D to reset the timer, and instructs the kernel to give control to the target program. As the program executes, each time a new basic block is entered a PC-time pair will be entered into the history buffer. When the stop breakpoint is reached, the kernel is reentered; it sends a message to D (and thence to the host) indicating the sampling is completed. The debugger then initiates the source history.

If the host were a single user workstation, with a sufficiently fast processor, it could absorb the functions of D.

2.4 Hardware Probe Method

This method relies on the "debugging co-processor" being able to detect (in parallel with instruction execution) when a new basic block is being executed by examining the value of a tag bit in a memory space parallel to that of the target. These tag bits would be set when the object program is loaded; the loader uses a table built by the compiler (and/or linker) containing the addresses of thoses instructions which begin new basic blocks. During execution, when an instruction is fetched whose corresponding tag is set, the value of that address and the value of a clock is saved in a large trace buffer. The extra memory and hardware needed could be implemented on the target configuration. However, the preferred scheme would be to keep this hardware separate, and implement the co-processor as part of an in-circuit emulator (ICE). The co-processor would consist of additional memory to represent the tags, the trace memory, a high resolution clock, and a simple processor to monitor the arrival of addresses and buffer their value along with that of the clock. The complete emulator would consist of a target processor, the co-processor, and additional control logic to enable the co-processor to control the target.

Existing ICEs usually provide a means of tracing references to specific addresses, bus cycles, or specific instructions, possibly qualified by a predicate. However, since the trace memories are generally small, and the set of predicates is limited to a small number, it is not feasible to use such emulators for tracing more than a small number of specific instructions or statements. The tag method proposed here provides a simple way to accomplish a more useful form of tracing over larger intervals, utilizing an ordinary memory of comparable bandwidth to the main memory. A tag scheme can also be used to implement an unlimited number of breakpoints on both data and program references, as proposed in [John82].

There is a problem when using the tag bit scheme for instruction tracing with architectures that implement instruction pre-fetch, such as the MC68000.[2] On such processors, the fetch of an instruction does not imply it will be executed. This situation would occur when the instruction following a branch has been fetched, but the branch is taken. On processors that use only single instruction pre-fetch, this problem could be handled by not recording an instruction address unless the next instruction is NOT tagged. This will work because the target of a branch is always tagged. The compiler need only guarantee each basic block has at least one untagged instruction, by inserting no-ops if necessary. (Estimates made from existing compilers show that such insertions would be needed only rarely). In future architectures, extra control lines could be provided to indicate which of several instructions in a prefetch queue have been discarded.

2.5 Comparison and Further Comments

The clear advantage of the hardware probe method is that it does not impose any execution overhead on the target program because the basic blocks are detected without the execution of additional instructions. The user can also change which code sections are traced (with no overhead associated with elided sections) without recompiling the source program. The software method is easier to try out initially.

The success of both methods naturally depends on having hardware that is fast enough to perform the real-time sampling. The fact that basic blocks are traced rather than every instruction or statement increases the chances that this is possible.

3.0 Current Work

3.1 Implementation

The real-time trace facility is currently being implemented for the programming language PL Modula

[2] MC68000 is a trademark of Motorola.

[Mah] targeted to an LSI-11[3] using the software probe method. The compiler runs on a VAX[4] under Berkeley UNIX. The sampling will be performed by a 68000 processor over a parallel interface, which will communicate with the VAX over a serial line initially, and later via an Ethernet connection. The language Modula was chosen as an initial vehicle because it is meant for process control applications, and because the in-house developed compiler was readily available.

During compilation, the compiler generates a pretty-printed version of the program source text, and a debug file mapping values of program addresses to the line numbers of corresponding basic blocks in the formatted file. After a trace sampling has occurred, and the debugger has received the sequence of PC-time ordered pairs, it determines the line numbers of the basic blocks executed and calls the display module to display them.

The debugger command processor and display modules have been written and debugged using simulated executions of Modula programs. The hardware will be installed and tested shortly, permitting actual user experience.

3.2 Editing Features

A large volume of information can be represented by the execution of a program, even over short intervals. A large part of this information may be uninteresting. For the trace to be useful, editing commands must be provided to allow the user to focus on the areas of interest. The trace should be presented in a convenient form for examination. Clearly a high-speed display is essential. In our implementation, the trace is presented in a style similar to the Cornell Program Synthesizer [Teit], with a marker that moves to the next (or previous) basic block executed rather than the statement, labeled by the time that block was executed. Additional commands allow positioning of the cursor to blocks selected by string matching or cursor movement commands, and a command to display the next execution time of the selected block. Additional commands exist to elide selected portions of the display, and to break out of loops or procedures and display the time the exit occurred.

4.0 Further Plans

If the trace scheme proves feasible, it will be incorporated into a production microprocessor software system for a Pascal-like process control language. The system would be built around a single user workstation to host the compiler and debugger, and would use the hardware probe method for doing the trace sampling. Rather than having the compiler produce a separate, pretty-printed version of the source text, the applications programmer would use a syntax directed editor (similar to [Teit]) to develop the formatted source text directly.

Acknowledgements

My thanks to D. Lorenzini and P. Rutter who read earlier drafts of this paper.

REFERENCES

[John82]
Johnson, Mark Scott, Some Requirements for Architectural Support of Software Debugging, Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices, April 1982.

[Kieb]
Kieburtz, Richard, William Barabash, Charles Hill, Stony Brook Pascal User's Manual, Dept. of Computer Science, SUNY at Stony Brook, 1979.

[Mah]
Mahjoub, Ahmed, A New Modula Compiler for the LSI-11, SIGPLAN Notices, June 1980.

[Satt]
Satterthwaite, Edwin H., Source Language Debugging Tools, PhD. Thesis, Computer Science Dept., Stanford University, (STAN-CS-75-494), 1975.

[Teit]
Teitelbaum, Tim, Thomas Reps, Susan Horwitz, The Why and Wherefore of the Cornell Program Synthesizer, SIGPLAN Notices, June 1981.

[3] LSI-11 is a trademark of Digital Equipment Corporation.

[4] VAX is a trademark of Digital Equipment Corporation.