A Software Instruction Counter

J. M. Mellor-Crummey and T. J. LeBlanc

Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Although several recent papers have proposed architectural support for program debugging and profiling, most processors do not yet provide even basic facilities, such as an instruction counter. As a result, system developers have been forced to invent software solutions. This paper describes our implementation of a software instruction counter for program debugging. We show that an instruction counter can be reasonably implemented in software, often with less than 10% execution overhead. Our experience suggests that a hardware instruction counter is not necessary for a practical implementation of watchpoints and reverse execution, however it will make program instrumentation much easier for the system developer.

1 Introduction

Several papers in previous proceedings have called for architectural support for program debugging and profiling [1, 8, 10]. This support typically includes a hardware instruction counter. Since most processors do not yet provide even this basic facility, system developers have been forced to invent software solutions.

A hardware instruction counter can be used for both profiling and debugging. It is particularly useful for tracing individual instructions, either to count the number of instructions executed between two given instructions or to implement reverse execution. However, most profiling and debugging functions do not actually require an instruction counter. When profiling, we usually wish to know the percentage of execution time spent in a particular loop or routine. The Unix utilities *prof* and *gprof* [6] provide reasonable results without the use of an instruction counter; they use periodic sampling to estimate the amount of time spent in a routine. When debugging, we are usually interested in the states of the computation, rather than individual instructions. Traditional cyclic debugging is based on repeated visits to the same set of states, gathering additional information about the states on each visit. An instruction counter is helpful because a state is uniquely determined by the number of instructions executed before the state is reached. However, other representations of state are also possible. This paper describes one such representation, called a *software instruction counter* (SIC).

There are several approaches that could be used to implement an SIC. The most obvious approach is to simulate a hardware instruction counter. That is, we could preface each instruction in a program with an instruction that increments a count in memory. This count will differ at most by one from the number of program instructions executed. Although an obvious implementation, it has an equally obvious drawback: both the code size and execution time would increase by a factor of two. A practical implementation must minimize both the number of instructions and memory references used to implement the instruction counter.

An alternative implementation might count only basic blocks. A software counter could be incremented upon entry to each basic block; the program counter would identify the precise location within a basic block. By augmenting a compiler to incorporate such an SIC in programs during compilation, we

This research was supported by NSF Research Grant No. CCR-8704492 and an ONR Young Investigator award, Contract No. N00014-87-K-0548.

could (a) take advantage of the knowledge of basic blocks already present in the compiler, (b) allocate a register to hold the software counter, and (c) localize the required instrumentation within the compiler.

The cost of a software instruction counter can be further reduced. Basic blocks are a static representation of the computation. A basic block represents a set of consecutive instructions that must be executed in sequence, however consecutive basic blocks may or may not be executed in sequence. We do not need to increment the instruction counter if a basic block falls through to the next basic block; we need only count branches to the start of a block. In fact, only *backward* branches and subroutine calls need be counted, since we cannot reuse a particular program counter value without branching backwards. Thus, a combination of the program counter and a count of the number of backward branches taken during an execution are sufficient to uniquely identify any state in the computation.

In the next section we describe the implementation of a software instruction counter based on this idea.

2 SIC Implementation

An instruction counter must be able to measure the progress of a program and interrupt the program after a certain amount of progress has been made [1]. To incorporate a software instruction counter within a program, we must modify the program, and any library routines it calls, to update a counter on each backward branch or subroutine call, and to test (perhaps implicitly) the value of the counter each time it changes value. When the counter reaches some predetermined value, a transfer of control to an appropriate handler must be arranged.

This program modification can best be performed by a compiler, since the information available during compilation is precisely what is necessary to add the appropriate code. However, considerable effort is required to augment a production compiler to instrument programs during compilation. Instead of modifying a compiler, we direct the compiler to generate assembly code, and use a separate program to instrument the assembly code before passing the code to the assembler. Despite the decoupling of program instrumentation from compilation, this approach achieves results nearly as good as those we would expect from a compiler-based instrumentation system, without requiring significant compiler modifications.

We use the GNU optimizing C compiler (*gcc* version 1.31) in our instrumentation system. One factor that influenced this choice is that *gcc* supports a command-line option that can direct the compiler to leave any of the machine registers unused by the code it produces. This option enables us to reserve a general-purpose register for counting branches. Without this capability, our SIC implementation would have to maintain a branch count in memory, at a considerable cost in performance. A second factor motivating the use of *gcc* is that it has a high quality optimizer. Optimization is important for accurately measuring the overhead of a software instruction counter; without optimization, overhead due to the SIC could be dominated by overhead due to poor code.

The second component of our instrumentation system is the *assembly code instrumentation program* (ACIP), which scans assembly code and instruments it to count backward jumps and subroutine calls. Since ACIP deals with assembly code, it is necessarily a machine-dependent part of our instrumentation system.

To simplify both ACIP and the code it produces, we take advantage of a simple observation: maintaining an exact count of the number of backward branches and subroutine calls is not necessary to make the SIC scheme work. During execution, it is permissible for the SIC to count both conditional branches that are not taken and forward branches. As long as the SIC increases monotonically and is consistently incremented *at least once* at each backward branch and subroutine call that occurs during program execution, a value of the PC will not be reused without the value of the SIC changing. With this condition satisfied, each (PC,SIC) pair uniquely identifies a particular instruction in the execution history of a program.

ACIP uses two passes over its assembly code input. In the first pass, ACIP identifies labels in the code and inserts each label along with its statement number into a symbol table. The symbol table handles global, local, and numeric labels, making it possible to use ACIP to instrument hand-generated assembly code as well as compiler-generated code. In the second pass, ACIP examines each statement to determine if it is a branch or a subroutine entry point.[1] All statements that do not fall into these two categories are echoed to the output unchanged. If the statement is a subroutine entry point (*gcc* sets up a frame pointer for each subroutine making entry points easy to recognize), code to count the subroutine call is added. If the statement is a branch, ACIP

[1]Changes in flow of control due to subroutine invocation can be counted at the point of call or inside the subroutine body. We chose to instrument each subroutine body since this strategy causes the least growth in code size.

makes a simplifying assumption: unless the target location of a branch is specified using a simple alpha-numeric label (not a PC-relative target, indirection through a register, or a target expression involving label arithmetic), the branch operation is assumed to be backward and is instrumented accordingly. This assumption may cause unnecessary instrumentation of some branches in the target program; however, without modifying the compiler to perform the instrumentation task or requiring ACIP to assemble its assembly code input and understand the semantics of jump tables, this assumption is unavoidable. For each branch whose target is a simple alpha-numeric label, the symbol table is queried about the location of the target. Forward branches are not instrumented, but backward branches and branches to locations defined externally are.

In the following sections, we assume that a register can be dedicated to support a software instruction counter.

2.1 SIC on a CISC

Implementing an SIC requires maintaining a counter value and transferring control to a handler when that count reaches a predetermined value. CISC processors, such as the VAX [2] and the Motorola 68020 [11], generally supply loop control primitives (e.g. decrement and branch) that can be used to efficiently implement an SIC. Sample code sequences to count backward branches and subroutine calls using these loop control instructions are shown for the 68020 and VAX in table 1. In each case, when the SIC register reaches -1, control transfers to the routine *SICregUflw*, which handles underflow of the SIC register. Unconditional backward branches and branches to unknown target locations are instrumented similarly.

On the 68020, we use the decrement-and-conditional-branch (*dbcc*) instruction to maintain a branch count in a register. The *dbcc* instruction takes three parameters: a condition code, a data register Dn, and a branch displacement. The semantics of the 68020 *dbcc* instruction are given in figure 1. The

```
if not $cc$ then
 Dn $\leftarrow$ Dn - 1;
 if Dn $\neq$ -1 then PC $\leftarrow$ PC + disp fi
fi
```

Figure 1: Semantics of the 68020 *dbcc* Instruction

68020 *dbcc* instruction does not alter the condition codes.

On the VAX, we use the subtract-one-and-branch instruction (*sobgeq*) to maintain a branch count in a register. The *sobgeq* instruction takes two arguments: an index operand, and a displacement. The semantics of *sobgeq* are shown in figure 2. The VAX *sobgeq*

```
index $\leftarrow$ index - 1;
if index $\geq$ 0 then PC $\leftarrow$ PC + disp fi
```

Figure 2: Semantics of the VAX *sobgeq* Instruction

instruction alters condition codes as part of its operation.

Although the loop control primitives on the 68020 and the VAX are similar, the 68020 *dbcc* is better suited to maintaining an SIC. As shown in table 1, the *dbcc* instruction folds the decrement of the SIC register in with the conditional branch. On the 68020, the *dbcc* costs the same number of cycles as a simple conditional branch, so SIC instrumentation introduces no overhead in the likely case where the branch is taken.[2] Since the VAX *sobgeq* instruction does not take a condition code as an argument, it cannot be used to directly replace a conditional branch. Therefore, the overhead of maintaining an SIC for a program on a VAX will be higher than on a 68020. Also, since the *sobgeq* instruction affects condition codes, use of it to maintain an SIC must either precede the computation of the condition codes for the branch, or be added at the branch target.

Without using these special loop control primitives on the VAX and 68020, the overhead of maintaining an SIC would increase. In both cases, it would be necessary to use the following instruction sequence to count a backward branch or subroutine call:

```
    dec register
    bgeq 1$
    jsr SICregUflw
1$:
```

In the average case, when the register doesn't underflow, this instruction sequence adds an overhead of two instructions to count a backward branch or a subroutine call. However, with the loop control instructions, the average case requires at most one instruction to update the SIC register.

In our implementation for the 68020, ACIP replaces conditional branches with a *dbcc*. Since conditional branches may use a 32-bit displacement on the 68020, replacing them with a *dbcc* may introduce an assembler error since *dbcc* only allows a 16-bit displacement. C programs typically consist of many

[2]Backward conditional branches are assumed to be taken more often than not, since most loops have more than one iteration.

Sequence Type	Original Code Sequence	Sequence With SIC Instrumentation	
		68020	VAX
Cond Branch	L1: ⋮ < *compute cc* > beq L1	L1: ⋮ < *compute cc* > dbne L1,d7 bne 1$ jsr SICregUflw 1$:	L1: ⋮ sobgeq r7,1$ jsr SICregUflw 1$: < *compute cc* > beq L1
Subroutine Entry	EntryPoint:	EntryPoint: dbra 1$,d7 jsr SICregUflw 1$:	EntryPoint: sobgeq r7,1$ jsr SICregUflw 1$:

Table 1: Sample Instruction Sequences for Implementing an SIC on a CISC processor

short functions, so it is unlikely that this assumption will cause problems. In practice, we have not encountered a program for which this assumption was violated.

2.2 SIC on a RISC

RISC processors support only simple instructions with the goal of having each instruction execute in a single cycle. Without complex primitives, such as the 68020's decrement-and-branch instruction (which enables SIC instrumentation of conditional branches with no overhead in the average case), counting each backward branch and subroutine call will cost at least one cycle to update the SIC register.

Surprisingly, examination of the instruction sets for three RISC processors reveals that the overhead for maintaining an SIC is not uniform across RISC processors. The HP Precision RISC processor [7] provides an add-to-immediate-and-trap-on-condition instruction (*ADDIT,cc*). We could use this instruction to add -1 to a register dedicated to the SIC and trap to the *SICregUflw* routine on underflow of the SIC register. Executing this instruction in the branch delay slot following each backward branch and at the entry point of each subroutine would properly maintain the SIC register.[3] On the other hand, processors such as SPARC [5, 13] and the MIPS R2000 [12] require a two instruction sequence to achieve the same effect. On the SPARC, we need an add instruction to update the count in the SIC register, followed by a conditional trap that transfers control to the *SICregUflw* routine. Similarly, on the R2000, we need an instruction to update the SIC register, followed by a conditional subroutine call to *SICregUflw*.

[3] Since branch delay slots for backward branches can be filled about 99% of the time [16], only about 1% of these updates to the SIC register add no cycles to the program execution.

2.3 Cost Experiments

In this section, we describe a series of measurements and predictions of the execution overhead that SIC instrumentation would add to each of a set of sample programs for both RISC and CISC processors. First, we describe measurements of SIC overhead for a set of programs executing on the CISC 68020 processor. Then, using our measurements of overhead on the 68020, we derive some predictions of the overhead of adding an SIC to programs executing on the SPARC and MIPS R2000 processors.

In measuring the overhead of an SIC, two types of overhead are important:

1. direct overhead that results from executing additional instructions to maintain the SIC, and
2. indirect overhead that results from making a register unavailable for program use by dedicating it to an SIC.

To measure both the direct and indirect overhead, we compiled each test program three different ways using *gcc* with optimization enabled. For the *baseline* version of the program, the compiler was permitted to allocate all of the machine registers to the program. For the *register* version, the compiler was directed to reserve one of the general-purpose machine registers, making it unavailable for use by the program. In the *count* version, ACIP instruments the *register* version of the program to use the reserved register to maintain an SIC.

By comparing the execution times of the *baseline* and the *register* versions of the program, we can measure the indirect cost of taking a register away from each program for use by the SIC. By comparing the *baseline* and the *count* versions of the program, we can measure the total overhead for the SIC. The difference between the *register* and *count* versions measures the direct overhead. Table 2 shows the measurements of the overhead for several sample programs executing on a 68020.

The impact of instrumentation overhead on the cost of subroutine calls is illustrated by measurement of the *Fibonacci* program shown in figure 3. Table 2

```
main(){ fib(34); }
fib(i)
int i;
{
    if (i <= 1) return 1;
    return fib(i - 1) + fib(i - 2);
}
```

Figure 3: The Fibonacci Test Program

shows that the instrumentation overhead for the Fibonacci program is 3.7%. The overhead due to maintaining an SIC is dwarfed by subroutine linkage, evaluation of the test, and computation of arguments for the recursive calls. We expect that typical subroutines have larger bodies than Fibonacci and therefore will likely incur less than 3.7% overhead for counting subroutine calls.

Execution timings for the compress program, a data compression utility, showed only a 0.2% overhead for maintaining an SIC. In comparing the *count* and *register* versions of this program, the *count* version, which maintains an SIC in a register, ran faster than the *register* version, which simply leaves a register unallocated. Possible explanations for this anomaly are that the addition of the SIC instrumentation to the compress code changed the page boundaries in the text segment resulting in a smaller working set, or that the new alignment of the instructions caused fewer collisions in the instruction cache.

The SIC overhead of 12% for the string-matching program grep when presented with the regular expression '[a-z]+Z' is the highest measured overhead for a real program and thus requires some explanation. *Advance*, a short procedure, is the heart of the matching algorithm for grep. For this test case, *advance* is called once for each of the 4×10^6 characters in the test data set. *Advance* consists of a switch statement in which the cases encountered for this particular regular expression consist of only a few instructions. Since *gcc* produces code that uses indirection through a jump table to dispatch the switch statement, ACIP treats the branch to an indirect target as potentially backward and adds SIC instrumentation. Compiler-based instrumentation of this code would recognize that indirection through this jump table is used exclusively for forward branches and omit the SIC instrumentation. To measure the overhead contributed by the unnecessary instrumentation of the switch statement, the switch statement instrumentation was removed manually; a subsequent test with the same regular expression took only 97.7 seconds on average to execute. Without the unnecessary instrumentation of the switch statement, the SIC overhead is only 5.9%, which is comparable to the instrumentation overhead measured for the calls to short subroutines in the Fibonacci test. In both cases, the instrumentation overhead is magnified because the number of instructions in the subroutine call is not large enough to dominate the cost of counting the subroutine call. The second test for the grep program uses a pattern which does not require invocation of the *advance* procedure. The absolute time of this test is much shorter, even though both tests use the same test data set. Note that for this case, the SIC overhead is only 1%.

Most of the 4.1% run-time overhead for the ditroff test is likely due to the cost of counting subroutine calls. An execution profile of the ditroff program (generated using *gprof* [6]) shows the execution time apportioned among a large number of calls to very short procedures. Also, in the ditroff test and the second grep test, the execution time of the program decreased when the compiler was given one less register to allocate to the program. Presumably, the compiler made a bad decision to keep an infrequently used value in a register, where the overhead of saving and restoring the register across subroutine calls outweighed the benefit of faster access to the value.

The final program tested was the Dhrystone 2.1 benchmark [17], which measures the integer performance of a compiler and machine pair. The execution overhead of adding an SIC to the Dhrystone was higher than all of the other programs measured. Most of this overhead is due to the very short procedures in the Dhrystone. We expect lower overhead for real programs since most programmers would use in-line procedures or macros for functions as short as those in the Dhrystone. Nonetheless, the results for the Dhrystone benchmark are useful because Dhrystone results are available for a wide range of machines. By determining the number of branches and subroutine calls counted in a single iteration through the Dhrystone, we can predict SIC overhead on a RISC using the Dhrystone figures reported for RISC processors.

test program instance	time in seconds			SIC overhead
	baseline[a]	register[b]	count[c]	
recursive Fibonacci	99.1	99.1	102.8	3.7%
compress	329.7	332.8	330.3	0.2%
grep '[a-z]+Z'	92.2	92.5	103.3	12%
grep 'ZZZ'	24.7	24.3	25.0	1%
ditroff	149.0	148.5	155.2	4.1%
lex	53.6	53.5	53.8	0%
Dhrystone 2.1[d]	108.4	108.4	122.0	12.5%

[a] All registers available (compiled with 'gcc -O').
[b] Register d7 unavailable (compiled with 'gcc -O -fixed-d7').
[c] SIC enabled (compiled with 'gcc -O -fixed-d7').
[d] Execution time for 500,000 Dhrystones.

Table 2: Measurement of Direct and Indirect Costs of SIC

For all sample programs, the indirect costs associated with sacrificing a register for branch counting were insignificant. Unless a compiler uses interprocedural register allocation, it typically will not utilize all of the registers inside procedure bodies; therefore, we expect that the indirect overhead for dedicating a register to an SIC will be small for most compilers. One shortcoming of the measurements that we performed was that we did not measure any programs using simulated floating point operations which typically are written in assembly code using all of the registers. In this case, the impact of allocating a register for the SIC would be greater. Even so, we found that in the Unix math subroutine library, only infrequently were all registers in use. This leads us to believe that the routines could be rewritten efficiently using one less register, even on a machine such as the 68020 which has only 8 general-purpose registers available.

Although we do not have direct measurements of the performance overhead of an SIC on a RISC, we can use published Dhrystone performance measurements of several RISC processors, coupled with our SIC measurements on a CISC, to develop performance projections for RISC architectures. Our method for predicting SIC overhead on a RISC is as follows:

1. We ran a Dhrystone benchmark on our 68020-based workstation for several different iteration counts. By noting the number of branches and subroutine calls counted for each test run of the benchmark program, the number of branches due to miscellaneous work other than the body of the benchmark was factored out and the number of branches for each execution of the Dhrystone was determined. We discovered that each Dhrystone executes 72 backward branches and subroutine calls that require update of the SIC.
2. For R2000 and SPARC processors, backward branches and subroutine calls can be counted using a two instruction sequence. This instruction sequence would need to be executed for each branch in a Dhrystone. Since each of the instructions in the sequence takes one cycle, the the total time spent on counting branches for each Dhrystone would cost 144 cycles.
3. Using the Dhrystone results and the cycle times listed for the R2000 and SPARC processors in the December 1988 Usenet distribution [15], the fractional overhead for SIC instrumentation of the Dhrystone on these architectures was estimated by comparing the execution time of 144 RISC instruction cycles for SIC instrumentation to the total cost of executing a single Dhrystone (which is computable from the Dhrystone benchmarks).

The derivation of our equation for the expected overhead of maintaining an SIC on a RISC processor is shown below.

$$overhead\ (\%) = \frac{time\ spent\ maintaining\ SIC}{single\ Dhrystone\ execution\ time}$$

$$overhead\ (\%) = \frac{\frac{updates}{Dhrystone} \times \frac{instr.}{update} \times \frac{cycles}{instr.} \times \frac{sec}{cycle}}{\frac{1}{Dhrystones\ per\ second}}$$

Replacing $\frac{updates}{Dhrystone}$ by 72 (measured value), $\frac{instr.}{update}$ by 2, and $\frac{cycles}{instr.}$ by 1, and simplifying we get:

$$overhead\ (\%) = \frac{144 \times Dhrystones\ per\ second}{clockrate\ (Hz)}$$

Using this equation, we computed overhead predictions for maintaining an SIC on three RISC computers for which Dhrystone information was available; table 3 summarizes these results. All of the Dhrystone numbers shown in table 3 reflect compilation with the -O3 optimization flag for each compiler. It is expected that these estimates of SIC overhead for the Dhrystone benchmark will be higher than the overhead encountered for real programs since the procedures in the Dhrystone benchmark are so small.

3 Debugging Using the SIC

We developed the software instruction counter to help us debug parallel programs using our debugging toolkit [4]. Our debugging methodology is based on cyclic debugging, in which the program is repeatedly executed, with each successive execution providing more detailed data about the execution path. Cyclic debugging is the most common technique used to debug sequential programs, however it does not always work for parallel programs. Successive runs of the same parallel program may take different execution paths, depending on the resolution of race conditions existing among processes. Thus, an execution of a parallel program is characterized not only by the program source code and input, but also by the relative order of interprocess events that occur during the execution.

In our approach to parallel program debugging, all interactions between processes are modeled as operations on shared objects [9]. During program execution each process records a history of its accesses to shared objects. Each time an object is read by a process, the object's *version number* is recorded in the process's execution history. When an object is modified, it generates a new version number, which is also recorded in the process's execution history. The union of the individual process histories specifies a partial order of accesses to each shared object. This partial order, together with the source code and input, characterizes an execution of the parallel program and is referred to as an *execution history*. We can *replay* an execution during the debugging cycle by using the execution history to enforce the same relative order on events, thereby providing repeatable execution of parallel programs.

Implicit in our approach is the assumption that a sequential process will always follow the same execution path when given the same input. This assumption is not true in the presence of an asynchronous transfer of control. Interrupts may occur at any time, causing a transfer of control to a completely different context. Asynchronous exceptions due to hardware failures and out-of-band messages have the same effect. In both cases, the state of shared variables and global resources may be affected by the exact state in which an interrupt or exception handler is invoked. An inability to reproduce the asynchronous transfer of control at precisely the correct moment makes it nearly impossible to debug these types of programs using traditional cyclic debugging techniques. At a recent workshop on parallel and distributed debugging, this problem was noted by the developers of other debugging systems [3, 14], none of whom had solved it.

To reproduce the effect of an asynchronous transfer of control, it is necessary to reproduce the transfer at precisely the same state in the computation. The program counter is not a sufficient indication of the state of a computation, since it describes a static location in the code segment, not a dynamic location in the execution path. The real-time clock is also insufficient, since it usually lacks the necessary resolution. A hardware instruction counter would be sufficient, but is not strictly necessary. Instead, a combination of program counter and software instruction counter can be used to describe exactly the state of a computation when an asynchronous transfer takes place.

To replay such programs, our replay mechanism must cause the transfer of control to occur in the state in successive executions. We can use the SIC to represent the state in which the transfer occurred. To record an asynchronous transfer of control in an execution history, we instrument all interrupt and exception handlers in the program so that they record the value of the program counter and the SIC at the time of the transfer, as well as an indication of the type of trap or interrupt. (Since we are dealing with shared memory programs, our monitoring is *intrusive* in that the programmer must insert the appropriate code in the program.) The additional overhead is approximately eight instructions per handler.

During program replay, the execution history is used to guide execution. If replay is enabled, the record of the next asynchronous transfer is read from the execution history, which contains the value of the SIC at the time of the original transfer. The SIC register is initialized to this value, causing *SICregUflw* to be invoked when the correct number of backward branches has occurred. At that time, a breakpoint is set at the location specified by the program counter value in the execution history record for the asynchronous transfer. Note that we do not require repeated executions of the breakpoint; we set the breakpoint only after we are certain that the next execution of the instruction of interest is the point of the trans-

System	Processor	Dhrystones	Clockrate	Predicted Overhead
Sun 4/260	SPARC	18048	16.67 MHz	15.6%
MIPS M/500	R2000	12806	8.00 MHz	23.1%
MIPS M/1000	R2000	22590	16.00 MHz	20.3%

Table 3: Overhead Predictions for Branch Counting on a RISC

fer. When the breakpoint occurs, we synthesize the trap or interrupt using the information stored in the history. Once a transfer has been synthesized, we reset the value in the SIC register to the time of the next asynchronous transfer and repeat the process.

The SIC can also be used to implement valuable debugging functions for a single process. In particular, the SIC can be used to implement watchpoints and reverse execution, without the hardware support previously thought to be required [1]. We can implement watchpoints by having the SIC serve the same function as the hardware instruction counter in [1]. The program execution is divided into intervals based on the value of the SIC. At the start of each interval the condition associated with the watchpoint is evaluated. Whenever the condition is met, the condition is known to occur during the previous interval. By dividing that interval into sub-intervals and restarting the process (possibly using checkpoints to avoid reexecuting the whole program), the SIC could be used to isolate the basic block where the condition is first satisfied. Single-stepping through this interval would provide the exact instruction responsible for the condition. A similar approach, based on reexecuting previous intervals, could be used to implement reverse execution. In both cases, the only advantage attained by hardware support is that the cost of maintaining the SIC (shown to be, on average, less than 10% of the total execution time) does not have to be paid.

4 Conclusions

Despite the lack of special hardware support, the software community has managed to provide low-cost tools for profiling and debugging. One problem that has eluded solution is the ability to recognize the exact state of a computation when a particular condition occurs, such as an asynchronous transfer of control or a watchpoint. We have described a reasonable-cost solution, based on a software instruction counter, that can be used to debug programs that allow asynchronous exceptions, and to provide watchpoints and reverse execution.

The cost metric for debuggers is typically assumed to be the effect on execution time. Our experience shows that by modifying compilers, assemblers, and system routines, we can implement the required functionality without an unreasonable impact on program performance. A hardware instruction counter could provide the same functionality without the need to modify system software, however. Unlike other proposed hardware features, such as the ability to trap on access or update of a specific data location [8], the tradeoff is not one of hardware cost versus function, but one of hardware cost versus software effort. For the case of the hardware instruction counter, this tradeoff should be taken into consideration by both hardware designers and software developers.

5 Acknowledgments

The authors would like to thank Steve Colwell, who performed some early experiments on branch counting and helped implement a preliminary version of the assembly code instrumentation program.

References

[1] T. Cargill and B. Locanthi. Cheap hardware support for software debugging and profiling. In *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–83, Palo Alto, CA, Oct. 1987.

[2] Digital Equipment Corporation. *VAX Architecture Handbook*. Digital Equipment Corporation, Maynard, MA, 1981.

[3] I. J. P. Elshoff. A distributed debugger for Amoeba. In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 1–10, Madison, WI, May 1988.

[4] R. Fowler, T. LeBlanc, and J. Mellor-Crummey. An integrated approach to parallel program

debugging and performance analysis on large-scale multiprocessors. In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, Madison, WI, May 1988.

[5] R. B. Gardner. SPARC scalable processor architecture. Sun*Technology*, 1(3):42–55, 1988.

[6] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *Proc. of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. SIGPLAN notices, Vol 17, No. 6, June 1982.

[7] Hewlett-Packard. *Precision architecture and Instruction Reference Manual.* Hewlett-Packard Company, Rockville, MD, 1987.

[8] M. Johnson. Some requirements for architectural support of software debugging. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 140–148, Palo Alto, CA, Mar. 1982.

[9] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[10] R. McLear, D. Scheibelhut, and E. Tammaru. Guidelines for creating a debuggable processor. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 100–106, Palo Alto, CA, Mar. 1982.

[11] Motorola. *68020 32-bit Microprocessor User's Manual, Second Edition.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[12] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riordan, C. Rowen, and D. Van't Hof. A CMOS RISC processor with integrated system functions. In *Proc. of the 1986 COMPCON.* IEEE, Mar. 1986.

[13] S. S. Muchnick. Optimizing compilers for SPARC. Sun*Technology*, 1(3):64–77, 1988.

[14] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, Madison, WI, May 1988.

[15] R. Richardson. Dhrystone 2.1 benchmark. Usenet Distribution, Dec. 1988.

[16] D. W. Wall and M. L. Powell. The Mahler experience: Using an intermediate language as the machine description. In *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 100–104, Palo Alto, CA, Oct. 1987.

[17] R. P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Notices*, pages 49–62, Aug. 1988.