Assembly-to-Assembly Translation for Instrumenting User Code

David W. Vorbrich

Department of Computer Science
Brown University
Providence, Rhode Island 02912

May 1994

This research project by David W. Vorbrich is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

May 1994

Date: 6-1-94

[illegible]
Robert H.B. Netzer

Assembly-to-Assembly Translation for Instrumenting User Code

David W. Vorbrich
Department of Computer Science
Brown University
Providence, RI, 02912-1910

E-mail: dwv@cs.brown.edu

The end result of this project is an assembly-to-assembly translator which automatically inserts instrumentation into SPARC code. The intent of this instrumentation is to provide a means by which memory references can be traced for future replay of a program's execution. The tool is general-purpose, allowing the user to link with a library of different hook functions for different tracing and replay algorithms. In this paper we present the design choices, inherent difficulties, and limitations involved in instrumenting assembly code and tracing memory references, along with a description of the tool itself.

1.0 Introduction

The ability to record a program's execution and replay it exactly in the future is an invaluable debugging aid. To record the execution, we *trace* information and save it into a *tracefile*. This file is later used during replay to provide details of the original execution (hereafter called execution), insuring the replay behaves exactly the same as the original. It has been shown that a replay is guaranteed to behave identically to the execution provided every read from memory made during the replay receives exactly the same value as it did during execution. [1] The tool implemented in this project supports this method of tracing.

We instrument the user's assembly code by inserting calls to hook functions provided by the user. These hooks are discussed in detail in the next section. The general idea is that for every event of importance to the trace, we call a hook function and provide it with the information necessary to record the event. Our instrumentation makes no assumptions about the hook functions themselves. Instead, we safeguard the user's original code so that the hooks may be modified frequently. This flexibility allows the user to try different tracing algorithms or correct errors in a particular method without having to re-instrument the original source code. Instead, the user simply rewrites the hooks, compiles them separately, and re-links the executable to be traced.

Before discussing the previous work on this project, we will first explain the tracing method being supported along with a description of the hook functions needed to perform the trace. Later we will discuss in detail the design choices faced during development and describe the SPARC instructions which can modify memory. The actual instrumenting tool and it's limitations are then discussed, followed by our test results and a conclusion.

2.0 Hook Functions and the Tracing Algorithm

Netzer and Weaver have developed a tracing algorithm which guarantees every read during replay receives the same value as during execution [1]. Not every read and write to memory need be traced, but for each memory reference the algorithm must check to determine if the current instruction needs to be saved into the trace file. By dividing the execution into *windows* of sequential instructions, the algorithm makes localized decisions based on recent memory references. The size of each window is determined at compile time by the user, and is controlled through the hooks the user provides. A software instruction counter (SIC, explained later) keeps track of when a new window should occur, and at each window boundary calls a hook function to initialize any information required by the next window. In order to support this method of tracing the user needs to provide each of the following hook functions.

- **_inithook**() is called to set up the tracing algorithm at the very beginning of execution. It is also responsible for setting the initial value of the SIC.
- **_exithook**() is called upon program exit, and should perform any final analysis and cleanup.
- **_windowhook**() is called at each window boundary and should reset state information to prepare for the next execution window. It is also responsible for resetting the value of the SIC.
- **_signalhook**() is called whenever a signal interrupts the execution so the trace algorithm is aware a signal occurred.

- **_systemhook**() is called prior to all system calls so the trace algorithm knows of the event.
- **_stsyshook**() **and _ldsyshook**() are called for stores and loads of memory which will occur inside a system call. Because the size of these operations varies with each instance, the size is passed as an argument along with the address in memory.
- **_ldhook**() **and _sthook**() are called for normal load and store instructions. The sizes of these instructions can be any of byte, halfword, word, or doubleword, so separate hooks for each are needed. The address of the reference is passed into these functions.
- **_restorehook**() is called after every restore instruction with a pointer to a buffer containing the values of all 16 registers which are modified by the restore.
- **_signalhook_special**() is called when a signal occurs during another hook function already in progress. This situation is tricky, as the order of events can get switched in the trace file. It is up to this hook to figure out the correct action to take to preserve the accuracy of the trace. This problem is discussed further in a later section.

To instrument a user's code for tracing we need to accomplish three things. We must call each of these hook functions for every corresponding event, provide each hook with the information necessary to perform the algorithm, and preserve the original functionality of the user's code. In the next section we discuss the work which has previously gone into solving this problem, followed by the many design issues which have led to our current implementation which meets the above requirements.

3.0 Prior Work and Motivation

The first attempt at inserting instrumentation into user code was made by Adam Stauffer in the summer of 1992. He directly modified the Gnu C compiler so that code to call the hook functions was emitted at the same time as the instruction itself. [2] He did this by editing the internals of gcc which are responsible for generating Gnu's intermediate representation of the code. The choice to instrument during this phase of the compilation was initially appealing, because "The modifications to the compiler are language independent and therefore any language added to the GNU language family is automatically instrumented." [2 pg 2] The Gnu compiler already supports many high level languages, and has been ported to almost every hardware platform. This flexibility made the Gnu C compiler a tempting choice to add instrumentation abilities.

An inherent problem within the C language, however, made this technique faulty. Arguments to function calls are ill defined in C.

> The order of evaluation of arguments is unspecified; take note that various compiler differ. [6 pg 202]

Due to this ambiguity of the C language, some expressions have no defined execution order. This makes it impossible for us to know in which order memory references within the expression will be executed. In the following example, we know the initial and final values of the variable ‘x’, but we have no guarantees on the input values for variables ‘a’ and ‘b’ because we do not know which will be evaluated first.

```
void foo (int a, int b);

void bar () {
    int x = 0;
    foo(++x,  ++x);
}
```

FIGURE 1. Example of argument ambiguity in C

Exact knowledge of the order of memory references is paramount to insure correct replay. This ambiguity makes it impossible to instrument the C language at a level close to the source representation.

A second effort to instrument user code was undertaken by Mitch Cherniack in the fall of 1993, and focused on the assembly representation of the user’s high-level code. [3] At this level, there is no ambiguity as to the execution order of statements, thus avoiding the idiosyncracies of C or any other high level language the user might have used. This method requires that the source code be translated into assembly before instrumentation can occur, but this functionality is easily provided by any compiler the user chooses.

At first, a translator was written in awk [5] to read in the original assembly representation of the user’s code, and emit the assembly containing the instrumentation to call hook functions. This initial tool worked for some simple tests, but it was evident that a more detailed and robust tool was required to handle complex programs and optimized code.

The fact that the tool almost worked, and its original results, indicated that the method of translating assembly-to-assembly to instrument the user's code was a promising one. It was especially appealing because the assembly representation of a user's program is so much more straighforward than the high level code, and it was also possible to insert the instrumentation without altering as complex a tool as the Gnu C compiler.

This project continues where the second translator left off. We chose to build a new translator in perl [7] instead of awk, because perl has better support for standard coding and calculations as well as the raw pattern-matching abilities of awk. Several bugs in the awk version of the translator have been solved, and much more design has gone into this version of the tool. The design issues faced, and the choices we made in implementing our tool are discussed in the next section.

4.0 Design Choices and Implementation

The design of our instrumenting tool revolved around two basic needs:

- Inserting calls to hook functions for *any* instruction which might be a memory reference
- Providing *all* information and functionality required by the tracing algorithm.

Locating memory references in assembly code is very straightforward, as most are just load or store commands (or versions thereof). The save and restore commands proved to be the only instructions which can reference memory that we could not trace, which we discuss in detail below. Providing hooks for window boundaries involved implementing a *software instruction counter* (SIC), and we also needed to call the initialization and cleanup hooks at the correct times. These requirements were obvious, but implementing them so that the program's functionality remained unchanged made them quite difficult. Each issue faced during our design of the tool, along with our implementation of the choices we made, is discussed in this section.

4.1 Registers

Almost every instruction in SPARC assembly involves the use of a register. The user's code is already using most registers, but our code, too, executes instructions which use registers. The SPARC architecture actually does us a favor in this case, however, as it leaves 3 global registers available for our use. Some other registers, however, are either inherently volatile, or are required by our instrumentation to pass arguments to the hook functions, and care needed to be taken to safeguard the original values in these registers.

4.1.1 Use of registers by our instrumentation

The instrumentation we add into a user's code requires the use of at least one scratch register. While it would be possible to store needed information in memory, and pull out what we need one at a time into this register, it would also be nice if more than one register were available for use. If we are to use any registers also used by the user's code, we must preserve the value in the register before our use, and subsequently restore it when we are finished. This would have been required of all registers we used, except for a very nice feature provided in the SPARC guidelines.

> The convention used by the SPARC Application Binary Interface (ABI) is that %g1 is assumed to be volatile across procedure calls, %g2...%g4 are reserved for use by the application program (for example, as global register variables), and %g5...%g7 are assumed to be nonvolatile and reserved for (as-yet-undefined) use by the execution environment. [4]

We thus assume that the global registers %g5, %g6, and %g7 are not used by the user's code, and are free for our own use. We scan the user's code during translation to make sure of this, reporting an error if we find these registers already in use.

As our design began to coalesce, we found that three values would be required continuously throughout our instrumentation, and would be referenced with high frequency. These three values are the SIC, the address in memory we set up for extra storage, and a value representing our *in-hook-flag*. Each of the available three global registers was assigned to one of these three values, which are discussed later in this section. One of the three, the *in-hook-flag*, is only needed periodically, so we can also use its register for temporary scratch space as needed.

4.1.2 Preservation of volatile registers in the user's code

There are some registers we could not avoid using, even if there were infinitely available global registers, simply because of specifications for the SPARC architecture. One such register, %o0, we needed to use to pass a single variable of information to each hook as it is called, while another, %g1, is written by the kernel when we perform a trap to obtain the value of the condition code (CC). In addition, %g1, %o7, and other registers are volatile across function calls, and must be protected even if their use is not required by our instrumentation or hooks.

We set up an area of memory within the data section of the program which we use as a virtual warehouse. Sometimes very little information needs to be stored there, and at other times the entire section is filled with information. The address of this area is stored in %g6, making it easy to access.

```
        Original Code                          Translated code
    <user code>                            set    .MYSTORAGE, %g6
    .                                      .
    .                                      .
    <user code>                            <user code>
                                           .
                                           .
                                           <user code>
                                           .
                                           .
                                           .seg "data"
                                           .align 4
                                       .MYSTORAGE:
                                           .word 0
                                           .
                                           .
                                           .word 0
```

FIGURE 2. Instrumentation storage area

We always store the same information at the same offset within this storage area, so storing and retrieving a value require only a single store or load instruction. We could also have saved our values on the user stack, using the stack pointer (SP) to access the area. We chose not to store values on the stack, however, for two reasons. First, we do not have our own reserved area of memory for each stack frame. Whenever we would store a value on the stack, we would have to increment the SP to prevent the user from walking on the same areas. Secondly, there are several areas which are reserved for each stack frame, but the size of this storage is not constant. For each new stack frame created, some area of memory is reserved for such things as arguments and register windows. Some of these areas are constantly sized, but others vary depending on the specifics of that stack frame. This ambiguity means we would not have an easy way of knowing where within the stack frame to begin our storage area. Some calculations per stack frame might be able to overcome this problem, but the fact that we would still have to use two instructions (a store followed by a modification of the SP) versus a single instruction to store values into the data section led us to chose the latter form of storage.

By utilizing this memory storage area, we are able to save the value of any register before we either use it or it becomes volatile, and then restore its original value when we are through with that section of instrumentation.

Original Code	Translated code
<user code>	<user code>
<user code>	st %o0,[%g6+8]
	<call hook function>
	<set up %o0 input to hook>
	ld [%g6+8],%o0
	<user code>

FIGURE 3. Protecting %o0 when entering a hook function

4.2 Supporting functionality for the tracing algorithm

The tracing algorithm this instrumentation is designed to support requires some functionality beyond calling hooks for every memory reference. Initialization and cleanup functions must be called at the execution's start and finish. In addition, much of the algorithm is based on dividing the execution into contiguous windows of dynamic instructions. A hook function is also required for each of these windows.

4.2.1 Initialization hook

Before the user's program begins, a hook must be called to initialize the algorithm and any data structures used within the hooks. Implementing this proved quite trivial, as we simply insert a call to this hook at the beginning of the user's **main**() routine.

Original Code	Translated code
main:	main:
<user code>	<save volatile registers>
.	call __inithook
.	nop
	<restore volatile registers>
	<user code>
	.
	.

FIGURE 4. Calling the initialization hook

If the C run-time library is used, this step can be taken during the start function. Another possibility would be to take advantage of the **.init** instruction of newer versions of the SPARC architecture. [10] Any code contained in this section will be executed before **main**() is executed.

4.2.2 Cleanup hook

When the program exits, some final analysis must be performed by the algorithm. We first provided this functionality in the same way as the initialization hook, by calling the cleanup hook at the very end of the **main**() routine. This method worked, provided the execution played straight through the code with no errors or purposely made calls to **exit**(). Considering the user is wanting to find errors in their code, the assumption that it will flow straight through the end of **main**() is an illogical one. Instead, we instrumented the **exit**() routine itself, inserting a call to the cleanup hook. Since **exit**() is also called when the **main**() routine is finished, it was no longer necessary to call the cleanup hook at the end of **main**(). We could also take advantage of newer versions of the SPARC architecture here, as we could for initialization. The **.fini** section is called after **main**() completes so we could insert any cleanup functionality at this point. [10]

4.2.3 Window hook

The tracing algorithm wants to create a new execution window for some dynamic constant number of events (usually instructions). In order to create the concept of a window, a hook function must be called at the beginning of each new window. Our instrumentation keeps a *software instruction counter* (SIC) which starts at a given limit, and calls the window hook when the counter reaches zero. The limit is provided by the user and is set during the initialization hook and reset each time the window hook is called. Our translator then instruments the assembly code to decrements the SIC at each event, and calls the hook when needed.

Keeping an actual count of instructions would either require hardware support or the addition of instrumentation between every assembly instruction to increment the counter. Since hardware support is not available on most machines, we must use a software approach. Incrementing the SIC with every instruction is both costly and unnecessary. Mellor-Crummey and LeBlanc proposed a method for maintaining a SIC which imposes much less overhead. [8] Instead of keeping a count of instructions, they keep a count of backward branches and procedure calls.

We have implemented this SIC to keep track of window boundaries. The user provides a constant value during the inithook, and when the SIC reaches this value a call is made to the window hook and the SIC is reset. While our SIC does not represent the actual number of instructions executed, the user can calculate an average number of instructions executed per branch or function call, and set their window size variable accordingly. In our implementation, the SIC is maintained in %g7. Since registers are 32 bits in size, one limitation we impose on the user is that the value given as the window size for the SIC must be less than 4 gigabytes.

4.3 Standard load and store instructions

In SPARC assembly, there are many different load and store instructions, but the use of each is very straightforward and easy to recognize. In our translator, we scan the code for any instruction which performs a load or store (or both). Upon finding one, we break down the expression representing the memory address being referenced and perform the calculation ourselves. The result of this calculation is stored in %o0 and passed to the appropriate hook function as the single input argument. When the hook returns, the original instruction is executed, and the program continues.

The calculation of the memory address is uniform across all load and store instructions. The size of the memory reference can be determined by the assembly instruction itself. Based on whether the instruction is accessing a byte, halfword, word, or double word, we call the appropriate sized hook function with the calculated address.

Original Code

```
<user code>
st      %o0,[%fp-20]
<user code>
```

Translated code

```
<user code>
<save volatile registers>
sub     %fp,20,%o0
call    __stwhook
nop
<restore volatile registers>
st      %o0,[%fp-20]
<user code>
```

FIGURE 5. Calculating memory address and calling hook for a store instruction

In the above figure, the store instruction is accessing memory at an offset of 20 from the current frame pointer (fp). To pass this information to the store hook, we insert a subtraction instruction to calculate the exact memory address, and store this value in %o0 to pass to the hook. The address could be a subtraction, as in this example, or it could be addition or a constant value as well. In these cases, the subtraction is replaced by an add or set instruction respectively.

4.4 Register windows

In SPARC architecture a concept known as *register windows* is used to maintain separate registers for different functions. Each function has it's own window, and when it calls another, a new register window is assigned to the new function. The total number of register windows is limited, however. Sometimes a new register window is needed, but none is available. At this time, the least-recently-used register window is dumped by the operating system to the user stack, and is then assigned to the new function. When the values of the old register window are needed, they are loaded back from the stack into the register window. New windows are assigned or relinquished via the save and restore commands, respectively. When a save occurs, it checks if a register window is available. If it is not available, it dumps the oldest register window to the stack and then resets the current register window. Likewise, the restore instruction either adjusts the window to the correct set of registers, or if they were dumped to the stack, it loads the register values from memory. For each new stack frame created, an area of the stack is reserved for a register dump should it become needed.

Because save and restore instructions do not access memory every time they are executed, they are much trickier to trace than standard load and store instructions. Due to their subtleties, each is discussed separately in this section.

4.4.1 The save instruction

When a save requires a register window to be dumped to the stack, it is the least-recently-used window which will be dumped. Each stack frame has a reserved area for register dumps, so the window being dumped will actually be stored in the stack frame of its original function, not in the current stack frame. This makes it easy to locate the saved register window for a particular frame, as it is stored in the same area on the stack as the function which needs it. This feature, however, introduces a very subtle problem which makes it impossible to trace save instructions accurately. If a save instruction is executed inside the kernel, it might cause a register dump of a user register window onto the user stack. This is actually very likely, because the least-recently-used window will have been in the user stack before entering the kernel.

Even though the kernel and user have their own respective stacks, through the save instruction it is very likely the kernel will modify the user stack. To see this phenomenon in action, we wrote a very simple program which is included as an appendix. In this program, we nested several functions on the stack, and then called a trap into the kernel. Prior to the trap we examine the register dump areas of the user stack, and do so again upon the return from the trap. Our test showed 5 different register dump areas had been modified during the trap into the kernel, indicating that 5 separate save instructions had modified the user stack.

This problem alone, however, is not enough to prevent us from tracing stores to memory caused during save instructions. We could perform a check of all user stack areas prior to every trap into the kernel, and again upon return, and trace the modified values. While potentially slow, this wouldn't occur too often and would accurately preserve memory. The bigger problem is posed by hardware interrupts. Hardware interrupts also cause a trap into the kernel, but they can occur at any time during the execution, between any two single assembly instructions. The brute force solution to this would seem to be obvious; simply check the user stack between every assembly instruction to see if an interrupt occurred which altered the stack. Even this broad a solution, however, will still not suffice. Because the interrupt can occur between any two instructions, it could occur between our memory check and the next instruction of the program. We would not detect the reference until after the next instruction during our next memory check, potentially causing two references to be out of order.

Luckily, the only areas of the user stack which will be modified by the kernel are those areas reserved for register dumps. We must make the assumption that the user's code will not read or write to these areas of the stack. Based on this assumption, we had two choices for our implementation:

- Since we assume the user will not access these restricted areas, and we are unable to trace them, we simply proceed with our instrumentation. If the user's code actually does access these areas of the stack, it is the programmer's error.
- We are unable to trace the memory in the restricted stack areas, however we can check the user's code to make sure the areas are not being referenced. This involves a costly check for each load and store operation, but assures the programmer that the replay provided by the trace is correct.

Accurate replay is our goal, so we chose to implement the second option. Beyond wanting to guarantee correct replay, the second option also reports the memory violations to the user. Although this aborts the replay, it does notify the user of a bug, thus making our implementation choice more useful to the programmer.

In our implementation we maintain a 2-level bit vector [9] to represent memory. When a new stack frame is created, we mark the restricted register area of the stack in our bit vector. When a stack frame is destroyed, we then unmark the bits indicating the area is no longer restricted. For each load and store instruction, we first call our own hook function which compares the address of the reference to our bitmap. If the address is in a restricted area, we report an error and dump core. If not, we let the execution proceed as normal. These actions are similar to the tracing algorithm used in the hook functions, and later we propose moving this memory checking functionality into the hooks to speed up performance.

4.4.2 The restore instruction

When a restore is executed, it destroys the current register window and restores the previous one. In doing this, it may or may not read the values from a previous register dump to the stack. Whether the values were on the stack, or still in the old window, the values contained in the registers will be modified by the restore, and the new register values will be needed during replay to guarantee identical execution. Thus, we must save the values of the modified registers for every restore instruction. Not all 32 registers are modified by the restore, as some are global and others overlap between windows. In the end, only 16 registers actually need to be traced to provide the correct values during replay.

In addition to tracing the values of modified registers, it is also necessary to perform some upkeep on our memory bit vector. When a restore occurs, a stack frame is destroyed. We thus unset the bits in our vector indicating the area is no longer restricted. This operation must be performed before the restore, while we still know the current SP. Dumping the register values, however, must be done following the restore, which is difficult if the restore appears in the delay slot of a function return. To avoid this, we chose to move the restore before the return, thus removing the delay slot optimization, but not altering the functionality of the program.

Original Code

```
<user code>
ret                      ! uses %i7 as targ
restore
```

Translated code

```
<user code>
st          %i7,[%g6+32]
<save volatile registers>
call        __unset_memtbl
mov         %sp,%o0
<restore volatile registers>
restore
st          %i0,[%g6+40]
st          %i1,[%g6+44]
.
.
st          %l7,[%g6+100]
<save volatile registers>
set         (.MYSTORAGE+40),%o0
call        __restorehook
nop
<restore volatile registers>
ld          [%g6+32],%o7
jmpl        %o7+8,%g0
nop
```

FIGURE 6. Unsetting protected memory area and tracing new register values

4.5 Maintaining condition codes

Condition codes are extremely volatile, and must be carefully preserved. To find the current value of the CC, we do a trap into the kernel, which writes the CC into %g1. Another possibility conceived by Shuang Ji, is to perform a series of tests and branches to determine and reset the CC, but we chose to use the trap in our implementation for simplicity. To preserve all original information before the trap, we first store the value of %g1 to our memory storage area. After this, we call the trap, and store the value of the CC into storage as well. Following whatever instrumentation we are performing, we simply reverse these steps to restore the CC and the original value of %g1.

Original Code

```
<user code>
<load instruction>
<user code>
```

Translated code

```
<user code>
st      %g1,[%g6+4]
t       0x20
st      %g1,[%g6+0]
<instrumentation>
ld      [%g6+0],%g1
t       0x21
ld      [%g6+4],%g1
<load instruction>
<user code>
```

FIGURE 7. Preserving %g1 and the CC when inserting instrumentation

While this is very straightforward, it is also quite expensive as it involves a trap into the kernel. To avoid this as much as possible, we perform some basic flow analysis to determine when the CC is volatile, and when we can safely ignore preserving its value.

Instead of blindly emitting instrumented assembly code, we maintain two buffers in our translator. One buffer contains the assembly which preserves CCs for all instrumentation, the other buffer does not. The base of the algorithm is simple. For any instruction which reads the CC, emit the buffer which preserves the CC prior to the read. If we encounter an instruction which writes to the CC, thus erasing it's previous value without reading it, we emit the buffer which does not preserve the CC. This basic approach must be modified, however, as control flow through basic blocks is unpredictable. Consider the following example:

```
1:  set CC
    jump 2


2:  straight line code with no branches
    and nothing that modifies the CC


3:  jump 4


4:  read CC
```

FIGURE 8. Example of tricky CC flow through basic blocks

In this example, the CC is set in block 1, and is not read until block 4. While we are instrumenting block 2 we have no idea if the CC is volatile or not. Because of this, we must always assume the CC is volatile inside a basic block until we encounter an instruction which writes a new value to the CC. If no such instruction is encountered before the end of the basic block or a branch instruction, we must assume the CC was volatile and emit the assembly which preserves its value. A more detailed data-flow analysis involving an extra pass or more through the code could overcome this assumption.

4.6 Signals

When signals occur, a signal handler is called to process the event. Signals can occur at any time, just like hardware interrupts, but unlike interrupts, the signal handler runs in user mode. This allows us to catch signals before they are processed, and call a hook function to record that a signal was received.

4.6.1 Catching signals

To catch signals before the handler takes over, we reset the signal handler for all signals at the very beginning of the user's execution. We also keep a record of what the old handler for each signal was, so we may later call the correct function. Once this is set up, our stub handler will be called for every signal that occurs. Inside our stub, we call the signal hook, and then check the value of the previous signal handler for this signal. If the old handler was a declared function, we simply call that function with the signal. If the old value was set to ignore the signal (SIG_IGN) we return from our function, otherwise the value was set to the default (SIG_DFL) and we take the appropriate action depending on what the signal was. Usually this involves exiting the program, but for some signals a core dump is also required.

Problems can arise when the signal occurs during a hook function which is already in progress. The hook is trying to record that one event occurred, and the signal will also record that a signal occurred. Depending on when the signal interrupts the hook, the record of these two events might be out of order. To catch this, before we enter a hook function we first set a specific flag (the *in-hook-flag*). Inside the signal handler we check the value of this flag. If it is not zero, the signal has occurred inside a hook, and the events might be traced out of order. Instead of calling the normal signal hook, we call a special signal hook which is only used in this situation. We provide the hook with the value of the flag, which uniquely identifies which hook function was interrupted, along with the SIC so the hook will know where in the current window we currently are. It is then up to the user to take the necessary action to insure these events are traced in the correct order.

Because we have no control over the hook function being interrupted, we have no way of knowing when it is dangerous for the signal to interrupt the function. The best we can currently do is to set our *in-hook-flag* immediately before calling the hook function (in the delay slot actually), and unset it upon return.

4.6.2 Catching resets of the signal handler

During the execution it is possible that the user will request that a new signal handler be installed for certain signals. If we ignore this, the new handler will be installed instead of our stub hook, and subsequent signals of that type will not be traced. Requests to change the signal handler are processed by the **signal**() function, so we have modified this function to meet our needs. Instead of changing the handler for the signal to the requested new handler, we leave our stub function as the handler. We do not ignore the new handler, however, but modify our own records of which function should be called by our stub when a signal is received. After the user calls **signal**() to reset the handler, when that signal type occurs our stub is called as always, but instead of calling the original handler for that signal, we now call the newly requested handler.

4.7 System calls

System calls provide the user with the ability to trap into the kernel to perform certain tasks. Most of these tasks involve memory references, which must be traced, but as before we have no way of tracing code inside the kernel. Luckily, in this case we have another option. Based on the system call id (SYSid) and the arguments provided, we can calculate what memory references to the user's address space will occur inside the trap to the kernel. In our implementation we have our own **system**() function which is similar in function to the signal handler discussed above, only it is not called automatically. When our translator comes across a trap into the kernel, it simply inserts a call to an internal hook function first. We calculate what addresses will be modified and how large the reference will be, and then call **_stsyshook**() or **_ldsyshook**() with the information to be traced. When we are done with our calculations, we return from our **system**() hook, and the execution will continue with the trap into the kernel.

4.8 Instrumenting delay slot instructions

In SPARC assembly the instruction immediately following some branch type instructions is executed even though the instruction appears after the branch. This type of instruction is called a delayed instruction, and it's location right after the branch is called the *delay slot*. Instructions which cause a transfer of control, and which also have this delay slot, are referred to as Delayed Control Transfer Instructions (DCTI). For many DCTIs, the delay slot is executed regardless of whether or not the branch is taken. If the instruction in the delay slot happens to be a load or store, it is sufficient in this case to insert the instrumentation before the DCTI as we know the memory reference will occur.

In some cases, however, whether or not the delayed instruction is executed depends on whether or not the conditional branch is taken. We cannot simply put our instrumentation before the DCTI, for if the delayed instruction is not executed, we would have falsely traced that it had. To get around this problem, we modify the assembly code to make it safe to instrument the delayed instruction, without the fear of false tracing.

In the following example, **bicc** represents a conditional branch who's delay slot will always be executed, and **bicc,a** will only execute its delay slot if the branch is taken. We use the symbol **~bicc** to indicate the logically opposite conditional branch, and **ba** is the branch-always command. We show two examples, one is a sequence of assembly with a delayed store which is always executed, and below is a conditionally executed delayed store. The assembly that we will translate this code in to is shown on the right of the figure.

Original Code	Translated code
12: bicc 40 16: store A 40: foobar	8: instrumentation 12: bicc 40 16: store A 40: foobar
12: bicc,a 40 16: store A 40: foobar	12: ~bicc,a ME 16: nop 20: instrumentation 24: store A 28: ba,a 40 ME: 40: foobar

FIGURE 9. Instrumenting delay slots and conditionally executed delay slots

4.9 Libraries

We have already discussed how we instrument the user's code to call the hook functions for our tracing algorithm, but we must go beyond this to get a true trace of the execution. Many function calls are to routines contained in system libraries. These library functions run in user space, and can be instrumented and traced just as the user's code can. We are thus able to trace all library calls by recompiling the source code for each library, instrumenting it with our tool during the process.

It is important to note that within our hook functions we do not want to call instrumented library routines, as we could end up in an endless loop. For example, if our hook function calls **printf()**, and the version we call is contained in an instrumented library, we might recursively call **printf()** forever. We maintain a small library of uninstrumented code for the functions required within our hooks.

4.10 Limited size immediate values

Some SPARC instructions take an immediate value limited to 13 bits in size as an argument (SIMM13s). Problems arise with these instructions when the SIMM13 is an expression containing two or more local labels. Because we are inserting numerous instructions into the user's code, the relative position of local labels previously in the code can change greatly. Prior to instrumentation, an expression containing one label minus another might have been within the SIMM13 size limit, but after our instrumentation this might no longer be true. To fix this problem, we extract the expression from the assembly instruction, and replace it with a scratch register. Then, before this instruction, we insert a command to store the result of the original expression in the scratch register. Registers can hold a full 32 bit value, so this removes the 13 bit limit.

Changing the position of the expression itself, however, causes its own difficulties. Many of these expressions contain the label '.' referring to the current line. When we move the expression to another line to place its value in a scratch register, care must be taken to add or subtract a proper constant to take into account the repositioning of '.'.

Original Code	Translated code
`<user code>`	`<user code>`
`add     %l2,(L3-.-4),%l2`	`sethi   %hi(L3-.-4-8),%g5`
`<user code>`	`or      %g5,%lo(L3-.-4-4),%g5`
	`add     %l2,%g5,%l2`
	`<user code>`

FIGURE 10. Rewriting expressions to use 32 bit register instead of immediate slot

5.0 Implementation Details of our Instrumenting Tool

As mentioned before, we chose to implement our translator in perl. [7]. Perl is a high-level programming language which is a super-set of awk, sed, grep, and other common system utilities. Perl is already in wide use, and should be available on most systems. If a user does not have perl already installed, it is easily obtainable via anonymous ftp. The code for our translator is included as an appendix, and the details of what each file does is left to a road map introducing that section. Here we will present the basic algorithm used to perform the translation to give an idea of what is happening when our tool is invoked.

The translation is broken into two main phases: pass 1 and pass 2. Before we begin the first pass, we first initialize some variables and load the supporting perl files we will need. We then prepare to read the user's assembly code through STDIN, open a temporary output work file, and finally emit the instrumented assembly through STDOUT.

Pass 1 prepares the user's assembly file for instrumentation by doing some initial tasks. Firstly, it scans the assembly to make sure the user's code does not make use of the global registers %g5-7 which are used by our instrumentation. We have found that the Sun C compiler, cc, makes use of %g5 in rare circumstances. Since we only use %g5 to represent our *in-hook-flag*, its value is only needed temporarily during instrumentation, not throughout the execution as the SIC and memory address are. If we detect the use of %g5 during pass1, we print a warning and set a flag telling pass2 to preserve the value in %g5 just like other volatile registers during instrumentation. For most assembly files this is not necessary, and we make free use of %g5 without saving its value. If we detect the use of %g6 or %7 during pass1, we report an error and abort.

The other primary action performed during pass 1 is to rewrite any DCTI expressions which contain conditionally executed delay slots. We use the translation discussed in section 3.7 to find an identical sequence of instructions which will allow safe and accurate instrumentation. The final action made in pass 1 is to emit the assembly necessary to set up our memory storage area. This area is marked with the label .MYSTORAGE, who's address is loaded into %g6 at the beginning of the execution. All of these modifications to the user's assembly code are written into a temporary output file which is then fed into pass 2 for further translation.

In pass 2 we perform all of the actions discussed in the last section. Primarily, we insert hook function calls for all memory references (including delayed instructions), maintain a bit vector of reserved memory areas on the stack, fix potentially too large immediate values in instructions, and perform CC flow analysis to reduce the number of system traps required to preserve the CC. All of these actions take place during a single pass through the temporary assembly file created in pass 1. Once we are finished instrumenting a sequence of assembly instructions, they are output to STDOUT.

6.0 Other Considerations and Limitations

While our translator will work for compiler generate assembly code, there are some situations which might arise in hand written code which our instrumentation might break. In addition, adding instrumentation to a user's code might incur a few unavoidable problems.

We have tried to pinpoint exactly what problems might arise, and discuss them in this section.

6.1 Use of libraries by the hook functions

The user must provide the hook functions, which we call from within our instrumentation and link with at compile time. Since we have no control over the user's code, it is up to the user to follow a few restrictions. When the user compiles their hook code, they must link statically with the system libraries. If not, the user's hooks will be calling the instrumented libraries which are linked with the original program to provide tracing. If the hooks call the instrumented libraries, which in turn call the hooks, we will get unpredictable results. Furthermore, the user must not use the **malloc()** routine within their hooks, or functions which use **malloc()**, like **printf()** and its siblings. The malloc routine is not re-enterable, making nested calls destructive. The user's hooks must not call **malloc()** because the hook might be called during a call to **malloc()** made by the original program. Due to this restriction, the user must use statically allocated memory, and make frequent use of the **write()** routine instead of **printf()**. If using **malloc()** is unavoidable, we have provided a shared memory version of alloc and free which are safe for use in the hooks.

6.2 Potential stack overflow

The stack size can only grow to a set limit. If the original program uses a stack which is close to this limit, but does not exceed it, adding calls to hook functions might cause an overflow error. This problem is unavoidable, but should not occur in ordinary programs. If this situation does arise, there is most likely a bug in the user's code which caused the stack to grow so large. The error encountered after adding instrumentation will potentially help locate the problem.

6.3 Label on a delay slot

While we have not encountered this situation in compiler generated assembly code, it is theoretically possible that a delayed instruction might be marked with a local label. This would allow a control flow to jump to the delay slot without executing the corresponding instrumentation if the delayed instruction was a memory reference. To catch such situations, new assembly instructions would need to be written to insure the proper instrumentation.

One possible method for dealing with labels on delay slots is to rewrite the instructions as shown in the following figure.

```
        Original Code                        Translated code
     instrumentation                      instrumentation
     bicc X                               bicc X
L1:                                  MY1:
     store A                              store A


     jump L1                              jump MY2


                                     MY2:
                                          instrumentation
                                          jump MY1
```

FIGURE 11. Translating labels on delay slots for proper instrumentation

When we encounter a label on a delay slot that needs to be instrumented, we change the label name to a newly created one, and later in the code we emit our own basic block beginning with a second newly created label which contains the needed instrumentation along with a jump back to the first label. While doing this, we keep a table which remembers we need to change all labels 'L1' with our own label, 'MY2' in this case. After pass 2 is complete, we can then make a 3rd pass and change all instances of 'L1' to 'MY2', insuring the proper instrumentation will be called if there is a jump to that delay slot.

6.4 Protection of our memory storage area

To perform our instrumentation, we have already shown the need for a memory location to temporarily store values we need to preserve. We set up our own private memory area during the translation phase, which the user's code has no knowledge of. If the user's code contains dangling pointers, however, it is possible it will reference our storage area. If the reference is a read from memory, our instrumentation will trace the event, and the value will be stored by the tracing algorithm if needed for replay. In this case, the memory reference is traced just like any other. If the memory reference is a write to this location, however, our stored data might become corrupt. If the user modifies our storage area the results are unpredictable. Some methods for dealing with this are discussed later in the enhancements section.

6.5 DCTI pairs

In SPARC assembly, some combinations of DCTI pairs are defined, but cause very unusual control flow, and should never occur in compiler generated code. It is also unlikely that they will ever occur in hand generated assembly unless the control flow is extremely intricate. In our implementation, if a DCTI pair is encountered, we notify the user and abort the translation. Here, we present a possible translation for dealing with this situation should it arise.

Due to the complexities of conditional branches, the SPARC manual clearly states the control flow for defined situations, and indicates all others are undefined (or machine dependent). [4 pg 54] The primary limitation stated is the first DCTI of a pair must be an unconditional branch. The second DCTI can be either an unconditional branch, a conditional branch, or an annulled conditional branch. Thus three possible DCTI pairs are possible, and the following figures show translations for each. Note that in all examples, the instruction at location 20 is never executed.

```
            Original Code                          Translated code
16 taken:        12, 16, 40, 60, 64...
16 not taken:    12, 16, 40, 44...

                                           12: ~bicc,a 40
12: ba 40                                  16: foobarB
16: bicc 60                                20: ba,a 60
20: foobarA                                24: foobarA

40: foobarB                                40: foobarB
44: foobarC                                44: foobarC

60: foobarD                                60: foobarD
64: foobarE                                64: foobarE
```

FIGURE 12. DCTI pair with conditional second instruction

Original Code	Translated code
16 taken: 12, 16, 40, 60, 64...	
16 not taken: 12, 16, 44, 48...	
	12: ~bicc,a 44
12: ba 40	16: foobarB
16: bicc,a 60	20: ba,a 60
20: foobarA	24: foobarA
40: foobarB	40: foobarB
44: foobarC	44: foobarC
60: foobarD	60: foobarD
64: foobarE	64: foobarE

FIGURE 13. DCTI pair with annulled conditional second instruction

Original Code	Translated code
16 always taken: 12, 16, 60, 64...	
12: ba 40	12: ba,a 60
16: ba,a 60	16: foobarA
20: foobarA	
	40: foobarB
40: foobarB	44: foobarC
44: foobarC	
	60: foobarD
60: foobarD	64: foobarE
64: foobarE	

FIGURE 14. DCTI pair with annulled unconditional second instruction

For each example, once the translation is performed, no instructions remain in a delay slot, making them safe to instrument.

7.0 Potential Enhancements

While functional, several enhancements could be made to the instrumenting tool to reduce its run-time overhead and refine its accuracy for detecting some problems.

7.1 Preserve only volatile global registers

Currently we save %g2-4 before calling a hook function, and restore them upon return. Global registers are volatile across function calls, and without knowledge of the hook functions we must preserve them to guarantee our instrumentation will work. It would be possible, however, to refine this by scanning the assembly code for the hook functions if it were available. If so, we could note which (if any) of the global registers are used within the hook function, and only preserve those specific registers.

7.2 Utilizing all unused registers

Many functions do not make use of all local registers available to them. If another pass were added to the translating algorithm, we could record every register that is not used within each specific function of the users code. Then, we could go back and store some of our values in these free registers during instrumentation instead of writing to our memory storage area. This still requires the same number of instructions, but a move from one register to another is faster than a store to memory, and would help bring down the overhead.

7.3 Manual control of the *in-hook-flag*

Currently we raise our *in-hook-flag* in the delay slot of the hook function call, and lower it immediately upon the function's return. If a signal occurs during the hook, we call a special signal hook as discussed earlier. Depending on where in the first hook the signal occurred, however, the user might want to take a different action. We could move the responsibility of raising and lowering the *in-hook-flag* into the hook function itself. The user could then position the flag to precisely the correct point where receiving a signal would result in an inaccurate trace.

7.4 Manual control of the memory bit vector

We maintain a 2 level bit vector and mark those memory areas which are reserved for register storage on the stack. Then, for each load or store instruction we check the address of the instruction against this vector to catch memory violations. This functionality is very similar to that of the tracing algorithm itself. It could be possible to set up two new hook functions, one for saves and one for restores, which would allow the user to set up and maintain the volatile memory bit vector. Then, for each load and store, the user could do the address check within the hook that is called. This would reduce the number of function calls made per memory reference by half.

8.0 Functionality Testing and Results

We tested our instrumentation on Sun SPARC 10 machines running SunOS 4.1.3. All instrumentation was done using the Sun C compiler (cc). We selected a test suite of three programs to instrument, each representing a different style of program.

- **gzip/gunzip** - is a file compression utility. It represents computationally intensive program executions.
- **gcc** - is the Gnu C compiler. It is a very large and complex program.
- **nethack** - is fairly large as well, but also represents interactive programs. It is a popular dungeon adventure computer game.

For each program, we compiled four separate test cases, linking with Sun libraries compiled with the same instrumentation as the test program. The four tests were:

1) **Uninstrumented** - is the program with no instrumentation.

2) CC **flow + manual registers** - includes our basic data flow analysis to reduce the number of traps needed to preserve CCs. We also manually save volatile local registers surrounding each block of instrumentation to our memory storage area.

3) CC **flow + save/restore** - includes our data flow analysis as before, but this time we surround each block of instrumentation with a set of save and restore instructions to preserve volatile local registers.

4) CC **always + save/restore** - does no data flow analysis, and always traps to obtain and reset the CC with every block of instrumentation. This test also includes the same save/restore method of preserving volatile local registers as in the last test.

The hook functions called by each of the three instrumented test cases were simply empty function declarations. This allows us to see the overhead of the instrumentation and hook function calls alone, without the added overhead that will be introduced by the tracing algorithm used. We ran each test 10 separate times and took the average user times of each to calculate the run times reported in figure 15. The window size used in all tests was 20 Meg. This figure is not that significant in our timing results, as the window hook is called so rarely in comparison to the other hooks that minor changes in its frequency have little effect on the overhead.

	gzip	gunzip	gcc	nethack
1) Uninstrumented run time	33.0	2.3	157.4	~30 min
2) CC flow + manual registers				
running time	1854.7	152.8	9270.9	~30 min
slowdown	56.2	66.4	58.9	not noticed
3) CC flow + save/restore				
running time	1811.7	149.7	10119.5	~30 min
slowdown	54.9	65.1	64.3	not noticed
4) CC always + save/restore				
running time	2669.4	187.9	10805.9	~30 min
slowdown	80.9	81.7	68.65	minimal

FIGURE 15. Overheads of varying instrumentation techniques

The overheads for tests #2 and #3 ranged from 55 to 66 times the uninstrumented versions, showing there is little difference between manually saving volatile registers during instrumentation vs. using the save and restore instructions. This fact will be significant, however, if the instrumentation is modified to pass more arguments to the hook functions. Currently, only one argument is passed to each hook, so we must save %o0 to pass the value. If other arguments are needed, more %o registers will need to be used. This will slow down test #2, but because test #3 uses the save/restore instructions provided by SPARC to save all local registers, its run time will remain unchanged.

In test #4, slowdowns ranging from 69 to 82 times the original execution were encountered. This clearly shows that performing data flow analysis to reduce the number of traps to preserve CCs is a win. Even better flow analysis could be performed if an additional pass were added to the translator, allowing us to keep track of volatile CCs across basic block boundaries.

For all tests, the performance of nethack was not noticeably slowed down. This was expected, as nethack is a user interactive game. Interactive programs typically have large periods of dead time while waiting for a user's action. This dead time far outweighs any slowdown introduced by our instrumentation, making the program change invisible to the user. In test #4 the game did seem to lag occasionally. This lag could have been due to temporary intense loads on our network, or may have been due to the slower instrumentation used in test #4. For most of the test, however, test #4 performed as quickly as all others.

	gzip	gunzip	gcc
1) Uninstrumented run time	33.0	2.3	157.4
2) CC flow + manual registers			
running time	701.1	98.6	4211.4
slowdown	21.3	42.9	26.8

FIGURE 16. Overheads for test #2 without stack memory correctness checking

To determine how much time was being taken up by our stack memory correctness functions, we recompiled our three non-interactive programs to execute test #2 without the added function calls. For these tests, we call the user's hook functions as before, but do not call the extra function before each hook to verify a valid memory access. Our results show that much of the execution overhead is incurred in these added functions. Without the memory checks, gzip ran 2.6 times faster than before, gunzip ran 1.6 times faster, and gcc ran 2.2 times faster. The functionality of the stack memory checks could be easily added to the user's hook functions. This would add to the overhead once again, but by removing the extra function call before each hook the run times would still remain faster than those reported in figure 15.

After running our tests with empty hooks to determine the run time overheads, we relinked the executables with our counting hooks. These hook functions increment individual counters for each hook function called, and print out the total counts upon exit. Figure 17 shows the total number of times each hook was called for each of the test cases. The signals received by nethack were manually generated, as nethack nicely catches all signals to make sure you really want to send them. This allowed us to send a ^Z and ^C during the game, without having to abort the execution.

	gzip	gunzip	gcc	nethack
_windowhook	3625	416	1373	223
_restorehook	3126535	7222	3972077	1552261
_signalhook	0	0	0	2
_ldbhook	104518603	10263717	14680968	5979585
_ldhhook	72506146	1876059	10112707	50715
_ldwhook	393130231	34695558	84533275	5177922
_lddhook	0	0	4	0
_stbhook	3944860	2355174	4005446	1101672
_sthhook	19268434	45757	725843	4618
_stwhook	153660417	13676930	39898398	1197169
_stdhook	0	0	18	0
_systemhook	206	173	5658	5804
_stsyshook	124	40	4912	2804
_ldsyshook	70	120	283	2910

FIGURE 17. Total number of times each hook was called

9.0 Conclusion

Dynamic tracing techniques show strong potential as future debugging aids, making their implementation desirable. Our compiler wrappers and translation code make it easy for the user to compile instrumented source files and link with tracing hook functions. Many subtleties of the SPARC architecture made instrumentation tricky, but our tool overcomes almost all of them. We found the SPARC save instruction impossible to instrument properly, so additional code was added to check all memory references and notify the user of any which might conflict with a register window save. This additional correctness checking could be easily moved into the hook functions themselves, lowering the run time overheads of the instrumentation. Our current instrumentation implementations introduced slow downs ranging from 55 to 82 times the originals, with average overheads around 60 times. Our results lead us to believe these overheads could be brought down by a factor of ~2 by shifting the responsibility of stack memory verification into the user hooks. The overheads could be further reduced by using more intricate data flow analysis to lessen the number of CC traps around instrumentation blocks.

Our instrumenting tool provides full support for current dynamic tracing techniques, allowing the development of more complete and faster tracing algorithms It is hoped through the use of this tool, a full tracing and replay tool will soon be realized.

10.0 Acknowledgments

Thanks go to Rob Netzer for his support in seeing this project through to its conclusion and Shuang Ji for his time saving advice and motivation. We also thank Katuya Tomioka for play testing all versions of nethack.

11.0 References

[1] Robert H.B. Netzer and Mark H. Weaver, "Optimal Tracing and Incremental Re-execution for Debugging Long-Running Programs," SIGPLAN '94 PLDI Conf. (June 1994).

[2] Adam Stauffer, "Instrumenting Variable References in GNU cc, Part 2," Final Project Submission (August 1992).

[3] Mitch Cherniack, "Issues in Instrumenting Compiler-Generate Code," Final Project Submission (December 1993).

[4] The SPARC Architecture Manual: Version 8, Sun Microsystems (1990).

[5] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, "The AWK Programming Language," Addison-Wesley Publishing Company (1988).

[6] B.W. Kernighan and D.M. Ritchie, "The C Programming Language," Prentice-Hall (1978).

[7] Larry Wall and Randal L. Schwartz, "Programming perl," O'Reilly & Associates, Inc. (1990).

[8] J.M. Mellor-Crummey and T.J. LeBlanc, "A Software Instruction Counter," Proc. of the Third ASPLOS (April 1989).

[9] Robert Wahbe, Steven Lucco, and Susan L. Graham, "Practical Data Breakpoints: Design and Implementation," SIGPLAN '93 PLDI Conf. (June 1993).

[10] Assembly Language Reference Manual for SPARC: Revision A, SunSoft of Sun Microsystems (Nov 1993).

12.0 Appendix A: Man Page

NAME
inst - translator for instrumenting assembly code

SYNOPSIS
```
inst < <sourcefile>.s > <targfile>.s
```

DESCRIPTION
inst will automatically instrument assembly code for execution tracing and future replay. Calls to hook functions which you provide are inserted into the original assembly file, allowing you to perform any tracing algorithm you have implemented. In order to ease the instrumenting phase of compilation, several wrappers have been developed for some compilers and assemblers. By putting the wrapper directory first in your path, they will be called at compilation, performing the instrumentation at the correct phase. (see Wrappers section)

USAGE
Using inst is quite simple assuming the hook code and user libraries are all correctly in place. Some environment variables can have an effect on the translation, and are discussed later. The main step required for using inst is to prepare a hook file.

To make a hook file, cd /u/rn/public/inst/hooks and copy one of the existing hook files into a new file. It is highly recommended that you begin with empty.c as a template. Everything that is required in the hook file is already contained in empty.c, so adding your algorithm or any other functionality to this file will insure a correct base. The hooks that must be declared are:

_inithook
: This function is called at the beginning of the execution and any initialization should be performed here. The function _setsic must be called specifying the size limit of the SIC.

_exithook
: This function is called on program exit, and should perform any cleanup operations.

_windowhook
: This function is called when the SIC (see Trace Windows section) reaches the given size limit. Inside this function the user must call _setsic providing the size limit so that the SIC can be reset.

_restorehook
: This function takes a pointer to a buffer containing the values of all 16 volatile registers during a

restore instruction.

_signalhook
: This function receives the SIC and also the signal type so that the user can record a signal took place, and use the SIC to determine the exact time.

_signalhook_special
: Similar to _signalhook except it is called when the signal has interrupted another hook in progress. This function also receives an extra argument indicating which specific hook was interrupted.

_ld[b,h,w,d]hook
: These four functions are called with an address indicating a load instruction is going to occur. The letters b, h, w, and d stand for byte, halfword, word, and doubleword loads respectively.

_st[b,h,w,d]hook
: As with the load hooks just before, only for store instructions.

_systemhook
: This function is called with the type of system call being made, so the user can record a system call occurred.

_stsyshook and _ldsyshook
: These two functions are called when load or store instructions will occur during a system call. Because memory operations are not restricted to bytes or words, separate hooks are required for these memory operations. Both the address and the size of the memory reference are passed to this function.

It is important that any variables declared within your hook file be declared as static to avoid name collisions with other files.

Once the hook code is ready, edit the Makefile to add a rule for your new file. It should be trivial to copy one of the other entries and rename the appropriate files. Once done, simply run make to build the object file and library for your hooks. Now you should *cd* ../*lib* and create a symbolic link to your newly created hook library. If you set your TFILE environment variable to the name of the new library, the next time you use the wrappers your hook library will be linked into the executable.

If you ever wish to change tracing algorithms or trace window sizes, re-instrumentation of the source code is not required. Instead, simply edit the hook file and relink the executable.

SUPPORT LIBRARY

The instrumentation requires several functions during execution, which are stored in a library in the /u/rn/public/inst/support directory. In addition, if system library functions are to be called from within your hooks, you must make sure the hook is calling an uninstrumented version of the library call. Within you hook code, call capitalized versions of all library calls. (for example, call OPEN instead of open) Now run 'nm' on the support library to make sure the functions you call from the hooks are already supported.

If not, you have a little hunting to do. It is helpful to use two windows during this procedure: one window should edit the Makefile in /u/rn/public/inst/support and the other window should be scanning the output of nm /usr/lib/libc.a (redirect to a file or less to view the output). Hunt through the nm listing for the definitions of the library calls you need to make. Add the file that contains them to the Makefile, and ALL functions defined or undefined within that file. Repeat this operation as necessary for any functions called from within the newly added library file until every object file has an entry in the Makefile and every function name is listed in that entry as arguments to convert.

After all of this, remove the file libhsupport.a and type make. All library files needed are extracted from /usr/lib/libc.a automatically, and then all named functions for that file are converted to upper-case letters via the /pro/aard/bin/sun4/convert utility. When the make finishes, run nm libhsupport.a to make sure it only contains functions declared in the instrumenting support code in /u/rn/public/inst/support/*.c or capitalized library function names.

The convert function is a program local to the Brown CS Department. If you do not have access to it, you may have to edit the sun source to rename the functions, and then rebuild your own local library.

Trace Windows

To support the concept of run-time trace windows, we have implemented a software instruction counter (SIC). The initial value of the SIC is set by your call to _setsic within the _inithook function. For every backward branch or function call, the SIC is decremented. If the new value has reached zero, we call your _windowhook function. Thus, it is vital that you also call _setsic within your _windowhook function if you wish trace windows to perform as they should.

COMPILING WITH WRAPPERS

The wrappers we provide are stub functions which we have set up to perform any necessary instrumentation during the normal compilation of a file. If you put /u/rn/public/inst/bin first in your path, the wrappers contained in that directory will be called when you compile a file. (Note: It is important you call your compiler by name, and not provide the full path. Some Makefiles contain explicit paths) We support the gcc, cc, and acc compilers with our wrappers.

In order for the wrappers to work correctly, it is necessary to have the environment variables TFILE and TLIB set correctly.

ENVIRONMENT VARIABLES

TFILE
: should contain the base name of the hook library you wish to link with. The library is already assumed to be in /u/rn/public/inst/lib. For example, if you wish to link with the counting hooks, set this variable to count.

TLIB
: contains the name of the C library you wish to link with. The current choices are vanilla, norm, save_res, and okill, which are tests #1-4 in the paper, respectively.

BUILDING_LIBC
: should be set only if you are compiling the C library source. If set to a non-null value it prevents linking with crt0 and the hook and support libraries.

ALWAYS_SR
: should be defined if you wish to force a save and restore around each block of instrumentation. The default is to do manual saves of volatile registers.

ALWAYS_CC
: should be defined if you wish to force a trap before and after each block of instrumentation to get and reset the condition code. The default is to perform our basic data-flow analysis to reduce the number of these traps.

The following environment variables were included for debugging purposes. For normal use, you do not need to define them.

NO_SLOTS
    if defined tells the translator to ignor potential instrumenting errors involving delay slots. The default is to report any DCTI pairs or labels on delay slots.

NO_SIC
    if defined disables the software instruction counter and thus all calls to _windowhook. The default is to leave the SIC on.

NO_STACK
    if defined disables the correctness checking performed to make sure the instrumented program is not reading from reserved register window areas of the stack. The default is to perform these checks.

NO_REGS
    if defined disables the translator from reporting register conflicts with the user's code. The default is to check for %g5-7 and report a warning if they are used. If only %g5 is used, we set a flag to make sure %g5 will be saved with the volatile registers during instrumentation. If %g6 or %g7 are used, the instrumentation will not work, and most likely the program will break.

TT_USEGCC
    over-rides whatever compiler was called, and forces gcc to be called. We have never needed to use this option, as the other wrappers seem to work perfectly well.

FILES

/u/rn/public/inst/	Base directory
+bin/inst	Actual translator
+bin/cc,gcc,as,etc	Wrappers for compilation
+perl/*.pl	Perl source of the translator
+hooks/*.[c,o]	User hook code and object files
+support/*	Code required by instrumentation
+lib/*.a	libc, crt0, hook, and support libraries

AUTHOR
    inst was written by:

        David W. Vorbrich (dwv@cs.brown.edu)

    in partial fulfillment of the Brown University Department of Computer Science Master's degree project under the guidance of Robert H.B. Netzer (rn@cs.brown.edu).

13.0 Appendix B: Road Map of Source Directory

The root of the source tree is located in **/u/rn/public/inst** and contains all of the source code, wrappers, hook files, supporting libraries, and documentation. Each directory contains a **README** file describing the files contained at that level. The general layout of the directories is as follows:

- **bin** - contains all wrapper functions along with the inst executable. This directory must appear first in your path in order for the wrappers to function properly.
- **docs** - contains this paper, the man page, and several data files used in compiling results for this paper.
- **hooks** - contains all user hook code source. The file empty.c is an excellent template for beginning a new hook file.
- **lib** - contains links to all supporting libraries required by the instrumented code. These include crt0.o, versions of libc.a, user hook files, and instrumentation specific functions.
- **perl** - contains all of the source code for the **inst** translator.
- **support** - contains support functions required by the instrumentation and hooks. These include some hand written files along with renamed functions extracted from the uninstrumented libc.a.

14.0 Appendix C: Hook File Template

```
/*
 This file contains simple hook functions showing all of the hooks
 necessary to trace code using our instrumenter. Each hook is
 listed in the following table along with its ID. This number
 uniquely identifies the hook function, and is used by the tracer to
 identify which hook it is currently executing should it happen to
 be interrupted by a signal.
*/

/*
 TABLE OF HOOK FUNCTION IDS

  1 inithook
  2 <not used>
  3 windowhook
  4 ldbhook
  5 ldhhook
  6 ldwhook
  7 lddhook
  8 stbhook
  9 sthhook
 10 stwhook
 11 stdhook
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "support_funcs.c"
#define WINSIZE (400*1024)

int tracefile;
int signalfile;

/*******************/
/** General Hooks **/
/*******************/

void _inithook () {
   tracefile = OPEN("TraceFile",O_WRONLY);
   signalfile = FOPEN("SignalFile",O_WRONLY);
   _setsic(WINSIZE);
}

void _windowhook () {
   _setsic(WINSIZE);
}

void _exithook () {
   WRITE(2, "Inside _exithook\n", 17);
   CLOSE(tracefile);
   CLOSE(signalfile);
}

void _restorehook(char *buf) {
}

/******************/
/** Signal Hooks **/
/******************/
```

```
void _signalhook(unsigned sic, int sig) {
}

void _signalhook_special(int hook, unsigned sic, int sig) {
}

/****************/
/** LOAD HOOKS **/
/****************/

void _ldbhook (void *add) {
}

void _ldhhook (void *add) {
}

void _ldwhook (void *add) {
}

void _lddhook (void *add) {
}

/*****************/
/** STORE HOOKS **/
/*****************/

void _stbhook (void *add) {
}

void _sthhook (void *add) {
}

void _stwhook (void *add) {
}

void _stdhook (void *add) {
}

/*********************/
/* SYSTEM CALL HOOKS */
/*********************/

void _systemhook (int sysid) {
}

void _stsyshook (unsigned add, unsigned sz) {
}

void _ldsyshook (unsigned add, unsigned sz) {
}
```

15.0 Appendix D: Wrapper Source Code

15.1 cc

```
#!/usr/local/bin/perl
# Script: cc
# Author: David W. Vorbrich <dwv@cs.brown.edu>

$ttpath = "/u/rn/public/inst/bin";
$cc = "/usr/bin/cc";

$#cmd = -1;

if ($ENV{'TT_USEGCC'})
{
 $ENV{'COMPILER_PATH'} = "$ttpath";
 @cmd = ("/cs/bin/gcc",@ARGV);
}
else
{
 @cmd = ($cc,"-Qpath",$ttpath,@ARGV);
}

#print "\x1b\x5b\x37\x6d",join(" ",@cmd),"\x1b\x5b\x6d\n";
exec(@cmd);
```

15.2 gcc

```
#!/usr/local/bin/perl
# Script: gcc
# Author: David W. Vorbrich <dwv@cs.brown.edu>

$ttpath = "/u/rn/public/inst/bin";
$cc = "/cs/bin/gcc";

 $ENV{'COMPILER_PATH'} = "$ttpath";
 $cmd = "/cs/bin/gcc @ARGV";

print "\x1b\x5b\x37\x6d$cmd\x1b\x5b\x6d\n";
exec("$cmd");
```

15.3 as

```
#!/usr/local/bin/perl
# Script: as
# Author: David W. Vorbrich <dwv@cs.brown.edu>

$args = "";
$as = "/usr/bin/as";

for ($i = 0; $i <= $#ARGV; $i++) {
   if ($ARGV[$i] !~ /^(.*)\.s$/ && $ARGV[$i] !~ /^-O/) {
$args .= " $ARGV[$i]";
    }
  }

$cmd  = "$as -S @ARGV";
# $cmd .= " | tee /tmp/tmpas.as";
$cmd .= " | /u/rn/public/inst/bin/inst";
# $cmd .= " | tee /tmp/tmpas.inst";
$cmd1 = " $as $args -";

if (($pid = fork()) == 0)
  {
   open(STDIN,"$cmd |");
   exec("$cmd1");
  }

waitpid($pid,0);
exit $?;
```

15.4 gas

```
#!/usr/local/bin/perl
# Script: gas
# Author: David W. Vorbrich <dwv@cs.brown.edu>

for ($i = 0; $i <= $#ARGV; $i++)
{
   if ($ARGV[$i] =~ /^(.*)\.s$/)
{
   die("instgas: multiple .s files\n") if ($base ne "");
   $base = $1;
}
   else
{
   $args .= " $ARGV[$i]";
}
}

die("instgas: no .s file specified.\n") if ($base eq "");

$cmd  = "/cs/bin/gas -S @ARGV";
$cmd .= " | /u/rn/public/inst/bin/inst";
$cmd .= " | /cs/bin/gas $args -";
exec("$cmd");
```

15.5 ld

```
#!/usr/local/bin/perl
# Script: ld
# Author: David W. Vorbrich <dwv@cs.brown.edu>

$libpath = "/u/rn/public/inst/lib";
$finallink = 0;
$tfile = $ENV{'TFILE'};
$tlib  = $ENV{'TLIB'};

for ($i = 0; $i <= $#ARGV; $i++)
{
 if ($ARGV[$i] eq "/usr/lib/crt0.o" || $ARGV[$i] eq "/lib/crt0.o")
   {
    $finallink = 1;
    $ARGV[$i] = "$libpath/crt0_inst.o" if (! $ENV{'BUILDING_LIBC'});
   }
 if ($ARGV[$i] eq "-lc")
   {
    $ARGV[$i] = "";
   }
}

$cmd = "/usr/bin/ld ";

if ($finallink && ! $ENV{'BUILDING_LIBC'})
{
 $cmd .= "-L$libpath @ARGV -lhsupport -l$tfile -lc_${tlib} -lhsupport";
}
else
{
 $cmd .= "@ARGV";
}

print "\x1b\x5b\x37\x6d**** $cmd\x1b\x5b\x6d\n";

exec($cmd);
```

16.0 Appendix E: Source for supporting functions

```
/*
 * File: support_funcs.c
 * Author: David W. Vorbrich <dwv@cs.brown.edu>
 */

int int2char(unsigned i, char *buf)
{
 int  j;
 char  mybuf[20];

 if (i == 0)
   {
     mybuf[0] = '0';
     j = 1;
   }
 else
   for (j = 0; i; i /= 10, j++)
     mybuf[j] = '0' + (i % 10);

 for (i = 0, --j; j; ++i, --j) {
     buf[i] = mybuf[j];
 }
 buf[i] = mybuf[j];
 buf[++i] = 0;
 return(i);
}


! global.s by David W. Vorbrich <dwv@cs.brown.edu>
! Functions to provide C code direct access to global registers
.text
.align 4
.global __getflag
.proc016
__getflag:
mov %g5,%o0
jmpl %o7+8,%g0
nop
.align 4
.global __getsic
.proc016
__getsic:
mov %g7,%o0
jmpl %o7+8,%g0
nop
.align 4
.global __setsic
.proc 016
__setsic:
mov %o0,%g7
jmpl %o7+8,%g0
nop
```

```
/*
 * File: malloc
 * Author: David W. Vorbrich <dwv@cs.brown.edu>
 */

#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

char *my_alloc(int size) {
   char *ret;
   int id;

   if ((id = SHMGET(IPC_PRIVATE,size,0700)) < 0)
return NULL;

   ret = SHMAT(id,NULL,0);
   SHMCTL(id,IPC_RMID,NULL);
   if ((int) ret == -1)
return NULL;

   return ret;
}

void my_free(char *p) {
   SHMDT(p);
}
```

```
/*
 * File: stack.c
 * Author: David W. Vorbrich <dwv@cs.brown.edu>
 *
 * Taken largely from tracing code written by Rob Netzer and modified
 * for use here.
 */

#define NULL 0

# definePAGESIZEBITS21
# definePAGEMASK((1<<PAGESIZEBITS)-1)
# definePAGESIZE(1<<PAGESIZEBITS)
# defineINDEXSIZE(1<<(32-PAGESIZEBITS))

static unsigned int masks[] = {
   0x00000001, 0x00000002, 0x00000004, 0x00000008,
   0x00000010, 0x00000020, 0x00000040, 0x00000080,
   0x00000100, 0x00000200, 0x00000400, 0x00000800,
   0x00001000, 0x00002000, 0x00004000, 0x00008000,
   0x00010000, 0x00020000, 0x00040000, 0x00080000,
   0x00100000, 0x00200000, 0x00400000, 0x00800000,
   0x01000000, 0x02000000, 0x04000000, 0x08000000,
   0x10000000, 0x20000000, 0x40000000, 0x80000000,
};

unsigned int *tbl[INDEXSIZE];

void _set_memtbl2(unsigned a) {
   unsigned int page, byte, byte_shift5, mask;

   page = (unsigned int)a>>PAGESIZEBITS;
   byte = (unsigned int)a & PAGEMASK;
   byte_shift5 = byte>>5;
   mask = masks[byte&0x1f];

   if (tbl[page] == NULL) {
tbl[page] = (unsigned int *)my_alloc(PAGESIZE/8);
BZERO(tbl[page], PAGESIZE/8);
   }
   tbl[page][byte_shift5] |= mask;
}

void _set_memtbl(unsigned a) {
   int i;
   for (i=0 ; i < 64; ++i,++a)
_set_memtbl2(a);
}

void _unset_memtbl2(unsigned a) {
   unsigned int page, byte, byte_shift5, mask;

   page = (unsigned int)a>>PAGESIZEBITS;
   byte = (unsigned int)a & PAGEMASK;
   byte_shift5 = byte>>5;
   mask = masks[byte&0x1f];

   tbl[page][byte_shift5] &= ~mask;
}

void _unset_memtbl(unsigned a) {
   int i;
   for (i=0 ; i < 64; ++i,++a)
_unset_memtbl2(a);
}
```

```
void _check_memtbl(unsigned a) {
   unsigned int page, byte, byte_shift5, mask;

   page = (unsigned int)a>>PAGESIZEBITS;
   byte = (unsigned int)a & PAGEMASK;
   byte_shift5 = byte>>5;
   mask = masks[byte&0x1f];

   if (tbl[page] != NULL && (tbl[page][byte_shift5] & mask)) {
WRITE(2, "===> PROGRAM READ FROM REGISTER AREA OF STACK <===\n", 51);
abort();
   }
}

void _delete_memtbl() {
   int i;
   for (i=0; i < INDEXSIZE; ++i) {
if (tbl[i] != NULL) my_free(tbl[i]);
   }
}
```

```
/*
 * File: mysighandler.c
 * Author: David W. Vorbrich <dwv@cs.brown.edu>
 * Created: Fri May 13 1994
 */

#include "mysignal.h"
#include <errno.h>
#include <stdio.h>

#define NUM_SIGS 31

extern int errno;
void (*mysig_table[NUM_SIGS + 1])();

void _mysig_handler(int sig, int code, struct sigcontext *scp, char *addr) {
   int pid, inhook, len;
   char buffer[80];
   unsigned sic;

   inhook = _getflag();
   sic = _getsic();
   if (inhook)_signalhook_special(inhook, sic, sig);
   else _signalhook(sic, sig);

   if (mysig_table[sig] == SIG_DFL)
switch (sig) {
case SIGHUP:
case SIGINT:
case SIGPIPE:
case SIGALRM:
case SIGTERM:
case SIGXCPU:
case SIGXFSZ:
case SIGVTALRM:
case SIGPROF:
case SIGUSR1:
case SIGUSR2:
   exit(1);
case SIGQUIT:
case SIGILL:
case SIGTRAP:
case SIGABRT:
case SIGEMT:
case SIGFPE:
case SIGBUS:
case SIGSEGV:
case SIGSYS:
case SIGLOST:
   SIGNAL(SIGABRT, SIG_DFL);
   exit(1);
case SIGURG:
case SIGCONT:
case SIGCHLD:
case SIGIO:
case SIGWINCH:
   return;
case SIGTSTP:
case SIGTTIN:
case SIGTTOU:
   pid = GETPID();
   KILL(pid, SIGSTOP);
   break;
}
```

```
  else if (mysig_table[sig] == SIG_IGN) return;

  else mysig_table[sig]();
}

void _mysig_setup() {
  int i;
  void (* tmp)();
  for (i = 1; i <= NUM_SIGS; ++i) {
tmp = SIGNAL(i, _mysig_handler);
if (tmp != -1) {
  mysig_table[i] = tmp;
}
  }
}

void *signal(int sig, void (*func)) {
  if (sig <= 0 || sig > NUM_SIGS || sig == SIGSTOP || sig == SIGKILL) {
errno = EINVAL;
return (-1);
  }
  mysig_table[sig] = func;
}


/*
 * File: mysystem.c
 * Author: David W. Vorbrich <dwv@cs.brown.edu>
 *
 * Taken largely from the vmon source code by Steve Reiss and modified
 * for use here.
 */

#define NULL 0
#define TRUE 1
#define FALSE 0
#define SYS_syscall 0

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/vfs.h>
#include <ustat.h>
#include <sys/uio.h>
#include <sys/utsname.h>

void _mysystem (int sysid,unsigned *args) {

  int i, write;
  unsigned address, size;

  _systemhook(sysid);

  if (sysid == SYS_syscall) {
sysid = *args;
args++;
  }

  for (i = 0; _mysystem2(i,sysid,args,&address,&size,&write); ++i) {
if (write) {
  _stsyshook(address,size);
```

```
}
else {
  _ldsyshook(address,size);
}
   }
}

int _mysystem2 (int cnt, int id, unsigned *args, unsigned *address,
      unsigned *size, int *write) {

  struct iovec *iov;
  int ct;

  *write = TRUE;

  if (cnt == 0) {
    switch (id) {
 default :
   return FALSE;
 case SYS_recvmsg :
 case SYS_sendmsg :
 case SYS_getmsg :
 case SYS_putmsg :
 case SYS_aioread :
 case SYS_aiowrite :
   return FALSE;
 case SYS_read :
 case SYS_getdents :
 case SYS_recv :
 case SYS_getdirentries :
 case SYS_recvfrom :
   *address = (int) args[1];
   *size = (int) args[2];
   break;
 case SYS_write :
 case SYS_send :
 case SYS_sendto :
   *address = (int) args[1];
   *size = (int) args[2];
   *write = FALSE;
   break;
 case SYS_stat :
 case SYS_lstat :
 case SYS_fstat :
   *address = (int) args[1];
   *size = sizeof(struct stat);
   break;
 case SYS_ioctl :
   *address = (int) args[2];
   *size = 64;
   *write = TRUE;
   break;
 case SYS_readlink :
   *address = (int) args[1];
   *size = (int) args[2];
   break;
 case SYS_gethostname :
   *address = (int) args[0];
   *size = (int) args[1];
   break;
 case SYS_fcntl :
   *address = (int) args[2];
   if (*address == NULL) return FALSE;
   *size = sizeof(struct flock);
   break;
```

```
case SYS_sigvec :
  *address = (int) args[2];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct sigvec);
  break;
case SYS_gettimeofday :
  *address = (int) args[0];
  *size = sizeof(struct timeval);
  if (*address == NULL) {
    *address = (int) args[1];
    if (*address == NULL) return FALSE;
    *size = sizeof(struct timezone);
  };
  break;
case SYS_getrusage :
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct rusage);
  break;
case SYS_getsockopt :
  *address = (int) args[3];
  *size = *((int *) args[4]);
  if (*address == NULL || *size == 0) return FALSE;
  break;
case SYS_getpeername :
case SYS_getsockname :
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = *((int *) args[2]);
  if (*address == NULL || *size == 0) return FALSE;
  break;
case SYS_getrlimit :
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct rlimit);
  break;
case SYS_statfs :
case SYS_fstatfs :
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct statfs);
  break;
case SYS_getdomainname :
  *address = (int) args[0];
  if (*address == NULL) return FALSE;
  *size = (int) args[1];
  break;
case SYS_ustat :
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct ustat);
  break;
case SYS_uname :
  *address = (int) args[0];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct utsname);
  break;
case SYS_readv :
  ct = (int) args[2];
  if (ct == 0) return FALSE;
  iov = (struct iovec *) args[1];
  *address = (int) iov->iov_base;
  *size = iov->iov_len;
  break;
case SYS_writev :
  ct = (int) args[2];
  if (ct == 0) return FALSE;
  iov = (struct iovec *) args[1];
  *address = (int) iov->iov_base;
  *size = iov->iov_len;
  *write = FALSE;
  break;
    }
  }
 else {
   switch (id) {
case SYS_gettimeofday :
  if (cnt > 1) return FALSE;
  *address = (int) args[0];
  if (*address == NULL) return FALSE;
  *address = (int) args[1];
  if (*address == NULL) return FALSE;
  *size = sizeof(struct timezone);
  break;
case SYS_getdirentries :
  if (cnt > 1) return FALSE;
  *address = (int) args[3];
  if (*address == NULL) return FALSE;
  *size = sizeof(long);
  break;
case SYS_readv :
  ct = (int) args[2];
  if (cnt >= ct) return FALSE;
  iov = (struct iovec *) args[1];
  iov += cnt;
  *address = (int) iov->iov_base;
  *size = iov->iov_len;
  break;
case SYS_writev :
  ct = (int) args[2];
  if (cnt >= ct) return FALSE;
  iov = (struct iovec *) args[1];
  iov += cnt;
  *address = (int) iov->iov_base;
  *size = iov->iov_len;
  *write = FALSE;
  break;
case SYS_recvfrom :
  if (cnt > 1) return FALSE;
  *address = (int) args[4];
  if (*address == NULL) return FALSE;
  *size = (int) args[5];
  break;
case SYS_sendto :
  if (cnt > 1) return FALSE;
  *address = (int) args[4];
  if (*address == NULL) return FALSE;
  *size = (int) args[5];
  *write = FALSE;
  break;
default :
  return FALSE;
    }
  }

  return TRUE;
}
```

17.0 Appendix F: Perl Source Code for inst

The source for inst is broken into seven separate perl files.

- **instrument.pl** - is the main driver of the program. It includes all other files, sets up the input and output files, and calls pass1 and pass2 in turn.
- **pass1.pl** - contains code for the initial pass over the user's code. During this phase we rewrite DCTI instructions with conditionally executed delay slots, check the user's code for global registers we reserve for instrumentation, and set up our memory storage area.
- **pass2.pl** - contains code for the primary pass over the user's code. During this phase we perform all instrumentation, inserting calls to the hook functions for all events required by the tracing algorithm.
- **funcs.pl** - contains all sub functions needed by pass1 and pass2 to keep track of state information and standard utilities.
- **hooks.pl** - contains sub functions for generating the assembly required to call each hook function.
- **rgxp.pl** - contains definitions for all regular expressions used to match the user's assembly file and trigger instrumentation output.
- **opposites.pl** - contains a single associative array indexed by DCTI's which provides the logically opposite instruction for each DCTI.

```
#!/usr/local/bin/perl
# Script: instrument
# Author: David W. Vorbrich <dwv@cs.brown.edu>

$DIR = "/u/rn/public/inst/perl";
require("${DIR}/rgxp.pl");
require("${DIR}/opposites.pl");
require("${DIR}/hooks.pl");
require("${DIR}/pass1.pl");
require("${DIR}/pass2.pl");
require("${DIR}/funcs.pl");
require("flush.pl");

$USE_SIC   = ! defined($ENV{'NO_SIC'});
$CHK_SLOTS = ! defined($ENV{'NO_SLOTS'});
$CHK_STACK = ! defined($ENV{'NO_STACK'});
$CHK_REGS  = ! defined($ENV{'NO_REGS'});
$ALWAYS_SR =   defined($ENV{'ALWAYS_SR'});
$ALWAYS_CC =   defined($ENV{'ALWAYS_CC'});

# TMP file is used as output from pass1 and read as input by pass2
$TMP = ".$$_pass1.s";

open(SRC, "<& STDIN");
open(TMP, "> $TMP");

&PASS_1();

&flush(TMP);
open(TMP, "$TMP");

&PASS_2();

unlink ($TMP);
```

```
#!/usr/local/bin/perl
# Script: pass1
# Author: David W. Vorbrich <dwv@cs.brown.edu>

sub PASS_1 {
  while(<SRC>) {

    # Skip blank lines and comments except !#PROLOGUE
    if ((! /^\s*$/ && ! /^\s*([!\#]|\/\*)/) || /($R_PRO)/) {

&PARSE_LINE();
if (/^_(\S+):/) {
  $FUNC = $1;
 }

# Check if the code being translated already uses global registers
# we depend on, if we encounter %g5, set a flag so we preserve
# its value during pass2
if (($CHK_REGS) && /($R_ILLEGAL)/ && ! /ascii/) {
  if (/%g5|%r5/) {
print STDERR "Used %g5... I will protect it: $_\n";
$SAVE_G5 = 1;
print TMP $_;
    }
  else {
die "Warning illegal register: $_\n";
    }
 }

# If we encounter a DCTI which always voids it's delay slot, put a
# nop there to avoid future hassels (like a label on the slot)
elsif ($W1 =~ /^($R_DCTInoslot)$/) {
  print TMP $_, "\t\tnop\t\t\t! Put nop in void delay slot\n";
 }

# If we encounter a DCTI who's delay slot may or may not be executed
# depending on if the branch is taken, rewrite the assembly so it
# is not ambiguous for translation
elsif ($W1 =~ /^($R_ba)|($R_cba)|($R_fba)$/) {
  $A = 1;
  $NEW_BRANCH = $OPPOSITE{$W1};
  ++$LABEL;
  $OLD = $_;
  $HEADER = sprintf
("\t\t${NEW_BRANCH} ${FUNC}.MINE${LABEL}\n\t\tnop\n\t\tba $W2\n");
  $FOOTER = sprintf("${FUNC}.MINE${LABEL}:\n");
 }

# If we see a ld/st instruction and we are in the delay slot of an
# annulled DCTI from above, then output rewritten assembly
elsif ($A == 1 && $W1 =~ /^(($R_LD)|($R_ST)|($R_CLR)|($R_LDST))$/) {
  print TMP $HEADER, $_, $FOOTER;
  $A = 0;
 }

# For all other instructions, if they were in the delay slot of an
# annulled DCTI first output the DCTI, then the current instruction,
# otherwise just print the current instruction
else {
  ($A == 1)?($A = 0, print TMP $OLD,$_):(print TMP $_);
 }
    }
  }
  &SETUP_STORAGE;
  }

# This function simply emits the code necessary for temporary storage in
# the data section of memory for saving and restoring register values we
# need to protect.
sub SETUP_STORAGE {
   print TMP "\t\t\t\t\t! MEMORY STORAGE FOR GLOBALS\n";
   print TMP ".seg\t\"data\"\n";
   print TMP "\t\t.align 4\n";
   print TMP ".MYSTORAGE:\n";
   for ($I = 0; $I < 40; ++$I) {
print TMP "\t\t.word 0\n";
    }
   print TMP "\t\t\t\t\t! END OF STORAGE AREA\n";

  }

1;
```

```
#!/usr/local/bin/perl
# Script: pass2
# Author: David W. Vorbrich <dwv@cs.brown.edu>

sub PASS_2 {
while(<TMP>) {
  $CUR = $_;


  &PARSE_LINE();# Break line into words

  &FIX_SIMM13();# Fix complex expressions in SIMM13's

  if (/^_(\S+):/) {
$FUNC = $1;
    }

  # If we see a new procedure definately print out the SIC code
  if (/($R_PRO)/) {
&MY_PRINT(0, $CUR);
     &INT_SICHOOK();
    }

  # If we see the start label, call the initialization
  elsif (/^($R_START)/) {
&MY_PRINT(0, $CUR);
&INT_INITHOOK();
    }

  # If we see a save instruction, mark our bitvector
  elsif ($W1 =~ /^($R_SAVE)$/) {
&PRINT_BUFS();
&FIX_SAVE();
if ($CHK_STACK) { &DO_SETMEM(); }
    }

  # If we see a restore (maybe in a delay slot) clear the bitvector
  # and dump the register values into the trace file
  elsif ($W1 =~ /^($R_REST)$/) {
if ($DCTI_SEEN) {
  &DO_RESTORE(1);# Actually eats the return statement
  $DCTI_BUF = "";
 }
else {
  &PRINT_BUFS();
  &DO_RESTORE(0);
 }
$DCTI_SEEN = 0;
$RETURN_TARG = "";
    }

  # If we see a DCTI, buffer info until we know when it's safe to
  # instrument.  Also, if it's a backward or conditional branch, increment
  # and check the SIC.
  elsif ($W1 =~ /^($R_DCTI)$/) {
if ($DCTI_SEEN != 1) {
  $DCTI_SEEN = 1;
 }
else {
  if ($CHK_PAIRS) {
die "========================> Branch in delay slot\n";
   }
  else {
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }
 }
if ($W1 =~ /^($R_RET)$/) {
  ($W2 !~ /^$/) ? ($RETURN_TARG = $W2) :($RETURN_TARG = "%i7");
 }

if ($BACK_BRANCHES{"${W2}:"} == 1) {
        &INT_SICHOOK();
 }
if ($W1 =~ /^($R_DCTInoslot)$/) {
  $NO_INST = 1;
 }
$DCTI_BUF = $CUR;
    }

  # For all instructions which touch memory, insert a call to the
  # appropriate hook function and clear up any delay slot buffering
  # from prior DCTI's.
  elsif ($W1 =~ /^($R_LD)$/) {
($NO_INST == 1) ? $NO_INST = 0 : &INT_LDHOOK1();
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }
  elsif ($W1 =~ /^($R_ST)$/) {
($NO_INST == 1) ? $NO_INST = 0 : &INT_STHOOK1();
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }
  elsif (/($R_CLR)/) {
($NO_INST == 1) ? $NO_INST = 0 : &INT_STHOOK2();
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }
  elsif ($W1 =~ /^($R_LDST)$/) {
($NO_INST == 1) ? $NO_INST = 0 : &INT_LDSTHOOK1();
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }

  # If we encounter a trap into the system, insert a call to the syscall
  # hook
  elsif (/($R_SYSCALL)/) {
&INT_SYSHOOK();
&MY_PRINT(0, $CUR);
    }

  # For every label encountered, record it's name to keep track of
  # backward branches.
  elsif ($W1 =~ /^($R_LABEL)$/) {
$BACK_BRANCHES{$W1} = 1;
if ($DCTI_SEEN == 1) {
  if ($CHK_SLOTS) {
die "=====================> Label on delay slot detected\n";
    }
  else {
&PRINT_BUFS();
    }
 }
&MY_PRINT(0, $CUR);
    }

  # The default action is to clear up our buffers and print out the
  # current instruction
  else {
```

```
&PRINT_BUFS();
&MY_PRINT(0, $CUR);
    }

  # This function actually emits the code once it knows if we need to
  # preserve the CCs or not.
  &CC_CHECK();

 }

# Simply here to catch the last buffer when the loop exits
($ALWAYS_CC) ? (print $OUTPUTcc) : (print $OUTPUT);

}

1;
```

```
#!/usr/local/bin/perl
# Script: funcs
# Author: David W. Vorbrich <dwv@cs.brown.edu>

# Doesn't actually print, but rather appends a string to one or more
# buffers based on a provided key.  1 = don't save CC,  2 = save CC
sub MY_PRINT {
   local ($WHICH, $STR) = @_;
   if ($WHICH == 1) {
$OUTPUT .= $STR;
    }
   elsif ($WHICH == 2) {
$OUTPUTcc .= $STR;
    }
   else {
$OUTPUT .= $STR;
$OUTPUTcc .= $STR;
    }
 }

# Actually only one buf, but the name stuck.  Here we append the buffer
# left over from a DCTI onto the actual output buffers.
sub PRINT_BUFS {
   if ($DCTI_SEEN == 1) {
&MY_PRINT(0, $DCTI_BUF);
$DCTI_BUF = "";
$DCTI_SEEN = 0;
$SLOT_SEEN = 1;
    }
 }

# Break the input line into seperate words
sub PARSE_LINE {
   (/^\s*(\S+)\s+/) ? ($W1 = $1) : ($W1 = "");
   (/^\s*\S+\s+(\S+)/) ? ($W2 = $1) : ($W2 = "");
   (/^\s*\S+\s+\S+\s+(\S+)/) ? ($W3 = $1) : ($W3 = "");
 }

# Split the arguments to a memory reference in order to determine what
# address it will be referencing.  These variables are referred to globally
# within other functions.  Namely, START_CALL().
sub GET_OPS {
   local($ARG) = @_;
   if ($ARG =~ /\+/) {
 $WH = index($ARG, "+");
 $O1 = substr($ARG, 1, $WH - 1);
 $OP = "add";
 $O2 = substr($ARG, $WH + 1, length($ARG) - $WH - 2);
 if ($O1 + 0 != 0) {
    $TEMP = $O1;
    $O1 = $O2;
    $O2 = $TEMP;
  }
    }
   elsif ($ARG =~ /-/) {
 $WH = index($ARG, "-");
 $O1 = substr($ARG, 1, $WH - 1);
 $OP = "sub";
 $O2 = substr($ARG, $WH + 1, length($ARG) - $WH - 2);
    }
   else {
 $O1 = substr($ARG, 1, length($ARG) - 2);
 $OP = "";
 $O2 = "";
```

```
    }
  }

# Trap to get the CC and put it into storage
sub SAVE_CC {
   &MY_PRINT(2, "\t\t\t0x20\t\t\t! Saving CC\n");
   &MY_PRINT(2, "\t\tst\t%g1,[%g6+0]\n");
  }

# Get a value from storage and trap to restore the CC
sub REST_CC {
   &MY_PRINT(2, "\t\tld\t[%g6+0],%g1\n");
   &MY_PRINT(2, "\t\t\t0x21\t\t\t! Restoring old CC\n");
  }

# Save volatile registers into storage.  Add other %oX registers if
# future hook functions take more than one argument.  Currently we
# only protect %o0.
sub SAVE_REGS {
   &MY_PRINT(0, "\t\tst\t%g1,[%g6+4]\t\t! Saving global registers\n");
   &MY_PRINT(0, "\t\tst\t%g2,[%g6+8]\n");
   &MY_PRINT(0, "\t\tst\t%g3,[%g6+12]\n");
   &MY_PRINT(0, "\t\tst\t%g4,[%g6+16]\n");
   if ($SAVE_G5 == 1) { &MY_PRINT(0, "\t\tst\t%g5,[%g6+20]\n"); }
   if ($ALWAYS_SR) {
&MY_PRINT(0, "\t\tsave\t%sp,-96,%sp\t\t! Saving non-global registers\n");
    }
   else {
&MY_PRINT(0, "\t\tst\t%o0,[%g6+32]\t\t! Saving non-global registers\n");
&MY_PRINT(0, "\t\tst\t%o7,[%g6+36]\n");
    }
  }


# Get values from storage and restore them to the correct registers
sub REST_REGS {
   &MY_PRINT(0, "\t\tld\t[%g6+4],%g1\t\t! Restoring global registers\n");
   &MY_PRINT(0, "\t\tld\t[%g6+8],%g2\n");
   &MY_PRINT(0, "\t\tld\t[%g6+12],%g3\n");
   &MY_PRINT(0, "\t\tld\t[%g6+16],%g4\n");
   if ($SAVE_G5 == 1) { &MY_PRINT(0, "\t\tld\t[%g6+20],%g5\n"); }
   if ($ALWAYS_SR) {
&MY_PRINT(0, "\t\trestore\t\t\t! Restoring non-global registers\n");
    }
   else {
&MY_PRINT(0, "\t\tld\t[%g6+32],%o0\t\t! Restoring non-global registers\n");
&MY_PRINT(0, "\t\tld\t[%g6+36],%o7\t\t\n");
    }
  }

# Emit a helpful comment and do any set up requried to call a hook
# function.  If we are passed a flag telling us this is a memory reference,
# also set up the correct argument to the hook.
sub START_CALL {
   local($CMT, $KND, $MEM_REF) = @_;
   &MY_PRINT(0, "\t\t\t\t! BEGIN INSTR-${CMT}: ${KND}\n");
   if ($MEM_REF == 1) {
if ($OP) {
   &MY_PRINT(0,"\t\t${OP}\t${O1},${O2},%g5\n");
  }
else {
   &MY_PRINT(0,"\t\tmov\t${O1},%g5\n");
  }
&MY_PRINT(0, "\t\tst\t%g5,[%g6+116]\n");
    }
   &SAVE_REGS();
   &SAVE_CC();
   if ($MEM_REF == 1) {
if ($CHK_STACK) { &DO_CHECKMEM(); }
&MY_PRINT(0, "\t\tld\t[%g6+116],%o0\n");
     }
 }



# Restore CCs and volatile registers and print out a closing comment
sub END_CALL {
   local($CMT, $KND) = @_;
   &REST_CC();
   &REST_REGS();
   &MY_PRINT(0, "\t\t\t\t\t! END INSTR-${CMT}: ${KND}\n");
 }

# Actually prints the output buffers. This is the only place real code
# is emitted. If we are in a delay slot, reading the CC, or just entered
# a new basic block, we must print out the buffer which preserves CCs.
# If we are writing a new value over the old CC, we can emit the prior
# buffers which do not protect the CC.
sub CC_CHECK {
   if ($SLOT_SEEN == 1 ||
$W1 =~ /^($R_LABEL)$/ ||
$W1 =~ /^($R_RDccplain)$/) {
print $OUTPUTcc;
$SLOT_SEEN = 0;
$OUTPUTcc = "";
$OUTPUT = "";
     }
   elsif ($W1 =~ /^($R_WRcc)$/) {
($ALWAYS_CC) ? (print $OUTPUTcc) : (print $OUTPUT);
$OUTPUTcc = "";
$OUTPUT = "";
     }
 }

# Store the values of all registers which will be modified by a restore
# instruction into a specified area of our storage.
sub DUMP_REGS {
   &MY_PRINT(0, "\t\tst\t%i0,[%g6+52]\n");
   &MY_PRINT(0, "\t\tst\t%i1,[%g6+56]\n");
   &MY_PRINT(0, "\t\tst\t%i2,[%g6+60]\n");
   &MY_PRINT(0, "\t\tst\t%i3,[%g6+64]\n");
   &MY_PRINT(0, "\t\tst\t%i4,[%g6+68]\n");
   &MY_PRINT(0, "\t\tst\t%i5,[%g6+72]\n");
   &MY_PRINT(0, "\t\tst\t%i6,[%g6+76]\n");
   &MY_PRINT(0, "\t\tst\t%i7,[%g6+80]\n");
   &MY_PRINT(0, "\t\tst\t%l0,[%g6+84]\n");
   &MY_PRINT(0, "\t\tst\t%l1,[%g6+88]\n");
   &MY_PRINT(0, "\t\tst\t%l2,[%g6+92]\n");
   &MY_PRINT(0, "\t\tst\t%l3,[%g6+96]\n");
   &MY_PRINT(0, "\t\tst\t%l4,[%g6+100]\n");
   &MY_PRINT(0, "\t\tst\t%l5,[%g6+104]\n");
   &MY_PRINT(0, "\t\tst\t%l6,[%g6+108]\n");
   &MY_PRINT(0, "\t\tst\t%l7,[%g6+112]\n");
 }

# More complex than it seems. First, unmark our bitvector before the
# restore, then print the restore itself. After this, dump the modified
# registers via DUMP_REGS() above, then call the restore hook providing
# a pointer into the storage area where the register values are buffered.
# After all of this is done, set up a jmpl to the original return target.
```

```
sub DO_RESTORE {
   local ($RET) = @_;
   if ($RET) {
&MY_PRINT(0,"\t\tst\t${RETURN_TARG},[%g6+120]\t\t! Save ret arg\n");
if ($CHK_STACK) { &DO_UNSETMEM(); }
     }
   &MY_PRINT(0, $CUR);
   &DUMP_REGS();
   &START_CALL("RESTORE_HOOK","restore",0);
   &MY_PRINT(0, "\t\tset\t(.MYSTORAGE+52),%o0\n");
   &MY_PRINT(0, "\t\tcall\t__restorehook,0\n\t\tnop\n");
   &END_CALL("RESTORE_HOOK","restore");
   if ($RET) {
&MY_PRINT(0, "\t\tld\t[%g6+120],%o7\t\t! Restore ret arg\n");
&MY_PRINT(0, "\t\tjmpl\t%o7+8,%g0\n"); # replaces return statement
&MY_PRINT(0, "\t\tnop\n");
     }
  }

# Pass the stack pointer to the set call to set our bitvector
sub DO_SETMEM {
   &START_CALL("SET_MEMTBL","_set_memtbl",0);
   &MY_PRINT(0, "\t\tcall\t__set_memtbl,0\n");
   &MY_PRINT(0, "\t\tmov\t%sp,%o0\n");
   &END_CALL("SET_MEMTBL","_set_memtbl");
  }

# Pass the stack pointer to the unset call to unset our bitvector
sub DO_UNSETMEM {
   &START_CALL("UNSET_MEMTBL","_unset_memtbl",0);
   &MY_PRINT(0, "\t\tcall\t__unset_memtbl,0\n");
   &MY_PRINT(0, "\t\tmov\t%sp,%o0\n");
   &END_CALL("UNSET_MEMTBL","_unset_memtbl");
  }

# Set up the address being referenced and then pass it to the check function
# to see if we are accessing a restricted area.
sub DO_CHECKMEM {
   &MY_PRINT(0, "\t\tld\t[%g6+116],%o0\n");
   &MY_PRINT(0, "\t\tcall\t__check_memtbl,0\t! Check if valid address\n");
   &MY_PRINT(0, "\t\tnop\n");
  }

sub FIX_SAVE {
   ($A, $B, $C) = split(/,/, $W2, 3);
   if ($B =~ /^(%[rglio][0123456789]+)$/) {
 &MY_PRINT(0, "\t\tsub\t${1},96,${1}\t\t! Insuring stack size\n");
 &MY_PRINT(0, $CUR);
     }
   else {
 $B .= "-96";
 $W2 = join(',', $A, $B, $C);
 $CUR = "\t\t" . $W1 . "\t" . $W2 . " " . $W3 . "\n";
 &MY_PRINT(0, $CUR);
     }
  }

# Looks ugly, but it's not that bad.  This function looks for parenthetical
# expressions.  If it finds one containing two local labels, then the
# result might be out of range if the expression is in a SIMM13 slot.
# Simply, the expression might be more than 13 bites, so we remove the
# expression in question, put it's result into a 32 bit register, and
# modify the original instruction to take this register.  We only do this
# for expressions containing two or more labels, and also take special
# care of the label '.' indicating current line, since we might move the
# line during instrumentation.
sub FIX_SIMM13 {
   return if /word/;
   return if /\s*sethi/;
   return if /\s*or/;
   return if /GLOBAL/;

   ($A, $B, $C, $D) = split(/,/, $W2, 4);

   if ($A =~ /\(.*($R_LABEL_REF).*[+-]($R_LABEL_REF).*\)/) {
$EXP = $A;
$W2 = join(‘,’, “%g5”, $B, $C, $D);
     }
   elsif ($B =~ /\(.*($R_LABEL_REF).*[+-]($R_LABEL_REF).*\)/) {
$EXP = $B;
$W2 = join(‘,’, $A, “%g5”, $C, $D);
     }
   elsif ($C =~ /\(.*($R_LABEL_REF).*[+-]($R_LABEL_REF).*\)/) {
$EXP = $C;
$W2 = join(‘,’, $A, $B, “%g5”, $D);
     }
   elsif ($D =~ /\(.*($R_LABEL_REF).*[+-]($R_LABEL_REF).*\)/) {
$EXP = $D;
$W2 = join(‘,’, $A, $B, $C, “%g5”);
     }

   if ($EXP) {
if ($W2 =~ /^(.*),+$/) {
   $W2 = $1;# Strip trailing extra commas
  }
$CUR = “\t\t” . $W1 . “\t” . $W2 . “ “ . $W3 . “\n”;

if ($DCTI_SEEN == 1) {
   if ($EXP =~ /(^\(\.[+-])|([+-]\.[+-])|([+-]\.\))/) {
($EXP1 = $EXP) =~ s/\)/-12\)/;
($EXP2 = $EXP) =~ s/\)/-8\)/;
     }
   else {
$EXP1 = $EXP2 = $EXP;
     }
   $NEW = sprintf(“\t\tsethi\t%%hi($EXP1),%%g5\n”);
   $NEW .=  sprintf(“\t\tor\t%%g5,%%lo($EXP2),%%g5\n”);
   $DCTI_BUF = $NEW . $DCTI_BUF;
  }
else {
   if ($EXP =~ /(^\(\.[+-])|([+-]\.[+-])|([+-]\.\))/) {
($EXP1 = $EXP) =~ s/\)/-8\)/;
($EXP2 = $EXP) =~ s/\)/-4\)/;
     }
   else {
$EXP1 = $EXP2 = $EXP;
     }
   $NEW = sprintf(“\t\tsethi\t%%hi$EXP1,%%g5\n”);
   $NEW .=  sprintf(“\t\tor\t%%g5,%%lo$EXP2,%%g5\n”);
   $CUR = $NEW . $CUR;
  }
     }
   else {
$EXP_SEEN = 0;
     }

   $EXP = $EXP1 = $EXP2 = $A = $B = $C = $D = $NEW = ““;
  }

 1;
```

```
#!/usr/local/bin/perl
# Script: int_hooks
# Author: David W. Vorbrich <dwv@cs.brown.edu>


sub INT_LDHOOK1 {
  $KIND = ($W1 =~ /^ld$/ ? "w" : substr($W1, length($W1) - 1));
  @T = split(/,/, $W2);
  &GET_OPS(@T[0]);
  &START_CALL("LOAD (1)", $KIND, 1);
  &MY_PRINT(0,"\t\tcall\t__ld${KIND}hook,0\n");
  &INT_HOOK_FLAG("ld", $KIND);
  &END_CALL("LOAD (1)", $KIND);
 }

sub INT_STHOOK1 {
  $KIND = ($W1 =~ /^st$/ ? "w" : substr($W1, length($W1) - 1));
  @T = split(/,/, $W2);
  &GET_OPS(@T[1]);
  &START_CALL("STORE (1)", $KIND, 1);
  &MY_PRINT(0,"\t\tcall\t__st${KIND}hook,0\n");
  &INT_HOOK_FLAG("st", $KIND);
  &END_CALL("STORE (1)", $KIND);
 }

sub INT_STHOOK2 {
  $KIND = ($W1 =~ /^clr$/ ? "w" : substr($W1, length($W1) - 1));
  &GET_OPS($W2);
  &START_CALL("STORE (2)", $KIND, 1);
  &MY_PRINT(0,"\t\tcall\t__st${KIND}hook,0\n");
  &INT_HOOK_FLAG("st", $KIND);
  &END_CALL("STORE (2)", $KIND);
 }

sub INT_LDSTHOOK1 {
  $KIND = ($W1 =~ /^swap$/ ? "w" : "b");
  @T = split(/,/, $W2);

  &GET_OPS(@T[0]);
  &START_CALL("LOAD/STORE (1)", $KIND, 1);
  &MY_PRINT(0,"\t\tcall\t__ld${KIND}hook,0\n");
  &INT_HOOK_FLAG("ld", $KIND);
  if ($OP) {
&MY_PRINT(0,"\t\t${OP} ${O1},${O2},%o0\n");
    }
  else {
&MY_PRINT(0,"\t\tmov ${O1},%o0\n");
    }
  &MY_PRINT(0,"\t\tcall\t__st${KIND}hook,0\n");
  &INT_HOOK_FLAG("st", $KIND);
  &END_CALL("LOAD/STORE (1)", $KIND);
 }

sub INT_INITHOOK {
  &MY_PRINT(0, "\t\tset\t.MYSTORAGE, %g6\t\t! Setting address of storage\n");
  &START_CALL("INIT_HOOK","init", 0);
  &MY_PRINT(0, "\t\tcall\t__mysig_setup,0\t\t! Setting up signal handler\n");
  &MY_PRINT(0, "\t\tnop\n");
  &MY_PRINT(0, "\t\tcall\t__inithook,0\t\t! INIT HOOKS\n");
  &INT_HOOK_FLAG("init");
  &END_CALL("INIT_HOOK", "init");
 }

sub INT_SYSHOOK {
    &MY_PRINT(0, "\t\t\t\t! Beginning of SYS_CALL hook\n");
    &MY_PRINT(0, "\t\tsave\t%sp,-96,%sp\n"); # -SA(MINFRAME) in asm_linkage.h
    &MY_PRINT(0, "\t\tst\t%g1,[%g6+0]\n");
    &MY_PRINT(0, "\t\tst\t%i0,[%fp+0x44]\n");
    &MY_PRINT(0, "\t\tst\t%i1,[%fp+0x48]\n");
    &MY_PRINT(0, "\t\tst\t%i2,[%fp+0x4c]\n");
    &MY_PRINT(0, "\t\tst\t%i3,[%fp+0x50]\n");
    &MY_PRINT(0, "\t\tst\t%i4,[%fp+0x54]\n");
    &MY_PRINT(0, "\t\tst\t%i5,[%fp+0x58]\n");
    &MY_PRINT(0, "\t\tmov\t%g1,%o0\n");
    &MY_PRINT(0, "\t\tadd\t%fp,0x44,%o1\n");
    &MY_PRINT(0, "\t\tcall\t__mysystem\n\t\tnop\n");
    &MY_PRINT(0, "\t\tld\t[%g6+0],%g1\n");
    &MY_PRINT(0, "\t\trestore\n");
    &MY_PRINT(0, "\t\t\t\t! End SYScall hook\n");
  }

sub INT_HOOK_FLAG {
    local($TYPE, $SIZE) = @_;
    if ($TYPE =~ /^st$/) {
if ($SIZE =~ /^b$/) {
    &MY_PRINT(0, "\t\tor\t%g0,8,%g5\t\t! Raising flag: 8\n");
  }
if ($SIZE =~ /^h$/) {
    &MY_PRINT(0, "\t\tor\t%g0,9,%g5\t\t! Raising flag: 9\n");
  }
if ($SIZE =~ /^w$/) {
    &MY_PRINT(0, "\t\tor\t%g0,10,%g5\t\t! Raising flag: 10\n");
  }
if ($SIZE =~ /^d$/) {
    &MY_PRINT(0, "\t\tor\t%g0,11,%g5\t\t! Raising flag: 11\n");
  }
      }
    elsif ($TYPE =~ /^ld$/) {
if ($SIZE =~ /^b$/) {
    &MY_PRINT(0, "\t\tor\t%g0,4,%g5\t\t! Raising flag: 4\n");
  }
if ($SIZE =~ /^h$/) {
    &MY_PRINT(0, "\t\tor\t%g0,5,%g5\t\t! Raising flag: 5\n");
  }
if ($SIZE =~ /^w$/) {
    &MY_PRINT(0, "\t\tor\t%g0,6,%g5\t\t! Raising flag: 6\n");
  }
if ($SIZE =~ /^d$/) {
    &MY_PRINT(0, "\t\tor\t%g0,7,%g5\t\t! Raising flag: 7\n");
  }
      }
    elsif ($TYPE =~ /^init$/) {
    &MY_PRINT(0, "\t\tor\t%g0,1,%g5\t\t! Raising flag: 1\n");
      }
    elsif ($TYPE =~ /^win$/) {
    &MY_PRINT(0, "\t\tor\t%g0,3,%g5\t\t! Raising flag: 3\n");
      }
    else {
die "Unknown in-hook-flag type received\n";
      }
    &MY_PRINT(0, "\t\tmov\t%g0,%g5\t\t\t! Lowering flag\n");
  }

sub INT_SICHOOK {
    if ($USE_SIC == 0) { return; }
    &MY_PRINT(0, "\t\t\t\t! Beginning of SIC code\n");
    &MY_PRINT(0, "\t\tst\t%g1,[%g6+32]\n");
    &SAVE_CC();
    &MY_PRINT(0, "\t\tsubcc\t%g7,1,%g7\n");
  ++$LABEL;
  &MY_PRINT(0, "\t\tbpos\t${FUNC}.MYSKIP${LABEL}\n");
  &MY_PRINT(0, "\t\tnop\n");
  &SAVE_REGS();
  &MY_PRINT(0, "\t\tcall\t__windowhook,0\n");
  &INT_HOOK_FLAG("win");
  &REST_REGS();
  &MY_PRINT(0, "${FUNC}.MYSKIP${LABEL}:\n");
  &REST_CC();
  &MY_PRINT(0, "\t\tld\t[%g6+32],%g1\n");
  &MY_PRINT(0, "\t\t\t\t! End of SIC code\n");
 }

1;
```

```
#!/usr/local/bin/perl
# Script: rgxp
# Author: David W. Vorbrich <dwv@cs.brown.edu>

# This file contains all regular expressions used throughout pass1 and
# pass2.  Names should be self explanitory, notes have been made where
# things are not quite obvious.

$R_ILLEGAL = “%g5|%g6|%g7|%r5|%r6|%r7”;
$R_SYSCALL = “(^\\s+)(t\\s+0|ta\\s+0|t\\s+%g0|ta\\s+%g0)”;

# The gcc source contains a line [ .ascii “!#PROLOGUE# 0” ] which is
# emitted during compilation.  We don’t want to trigger our expression
# on this comment, but only on the real PROLOGUE comments... thus this
# expression _must_ be anchored to beginning of line.
$R_PRO       = “^\\s*!#PROLOGUE# 0”;

#$R_START      = “start:”;
$R_START     = “_main:”;
$R_RET       = “(ret|jmpl|jmpl|retl|rett)”;
$R_CALL      = “call”;
$R_LD        = “ld(sb|sh|ub|uh|d)?”;
$R_ST        = “st(b|ub|sb|h|uh|sh|d)?”;
$R_CLR       = “clr \\[|clrb|clrh”;
$R_LDST      = “swap|ldstub”;
$R_SAVE      = “save”;
$R_REST      = “restore”;

$R_b         = “b(ne|nz|e|z|g|le|ge|l|gu|leu|cc|geu|cs|lu|pos|neg|vc|vs)”;
$R_ba        = “($R_b),a”;
$R_bnocc     = “b(n|a|n,a|a,a|,a)?”;
$R_Bicc      = “($R_b)|($R_ba)|($R_bnocc)”;

$R_cb        = “cb(3|2|23|1|13|12|123|0|03|02|023|01|013|012)”;
$R_cba       = “($R_cb),a”;
$R_cbnocc    = “cb(n|a|n,a|a,a)”;
$R_CBccc     = “($R_cb)|($R_cba)|($R_cbnocc)”;

$R_fb        = “fb(u|g|ug|l|ul|lg|ne|nz|e|z|ue|ge|uge|le|ule|o)”;
$R_fba       = “($R_fb),a”;
$R_fbnocc    = “fb(n|a|n,a|a,a)”;
$R_FBfcc     = “($R_fb)|($R_fba)|($R_fbnocc)”;

$R_t         = “t(ne|nz|e|z|g|le|ge|l|gu|leu|lu|cc|geu|cs|pos|neg|vc|vs)”;
$R_tnocc     = “t(n|a)?”;
$R_Ticc      = “($R_t)|($R_tnocc)”;

$R_WRcc      = “cmp|tst|btst|cpop2|([a-z][a-z]+cc[a-z]*)”;
$R_RDccbranch = “($R_b)|($R_ba)|($R_cb)|($R_cba)|($R_fb)|($R_fba)|($R_t)”;
$R_RDccplain  = “addx|addxcc|subx|subxcc”;
$R_RDcc      = “($R_RDccbranch)|($R_RDccplain)”;

$R_UDCTI     = “($R_CALL)|($R_RET)”;
$R_DCTI      = “($R_UDCTI)|($R_Bicc)|($R_CBccc)|($R_FBfcc)”;

$R_CTI       = “($R_Ticc)|($R_DCTI)”;

# This expression represents branches which are unconditional, and always
# annul their delay slot
$R_DCTInoslot = “bn\,a|fbn\,a|cbn\,a|ba\,a|b\,a|fba\,a|cba\,a”;

$R_LABEL_REF  = “[a-zA-Z_\$.][a-zA-Z_\$.0-9]*”;
$R_LABEL      = “[a-zA-Z_\$.][a-zA-Z_\$.0-9]*:”;
```