RYO: A VERSATILE INSTRUCTION INSTRUMENTATION TOOL FOR PA-RISC

Daniel F. Zucker and Alan H. Karp

Technical Report: CSL-TR-95-658

January 1995

This work was supported by NASA under contract NAG2-842 and Hewlett-Packard under gift No. 23487.

RYO: a Versatile Instruction Instrumentation Tool for PA-RISC

by

Daniel F. Zucker and Alan H. Karp[1]

Technical Report: CSL-TR-95-658

January 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

RYO (Roll Your Own) is actually a family of novel instrumentation tools for the PA-RISC family of processors. Relatively simple awk scripts, these tools instrument PA-RISC assembly instruction sequences by replacing individual machine instructions with calls to user written routines. Examples are presented showing how to generate address traces by replacing memory instructions, and how to analyze floating point arithmetic by replacing floating point instructions. This paper introduces the overall structure and design of RYO, as well as giving detailed instructions on its use.

Key Words and Phrases: instrumentation, address tracing, cache, arithmetic

[1] Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94306

Copyright © 1995
by
Daniel F. Zucker and Alan H. Karp

Contents

1 What Is RYO **1**

1.1 Overview 1

1.2 Advantages and Disadvantages 1

1.3 Implementation 3

1.3.1 Branch Delay Slot 3

1.3.2 Saving and Restoring Machine State and Parameter Passing 6

2 How to Use RYO **7**

2.1 Compiler Errors 7

2.1.1 Branch Targets 8

2.1.2 Indirect procedure calls 8

2.2 Long Branches 8

2.3 Offset.awk 9

3 Example Data **9**

3.1 Arithmetic 10

3.2 Memory 10

4 Conclusion **11**

5 Acknowledgments **11**

A Source Code for Ryols.awk **12**

B An Example Instrumentation Routine **25**

List of Figures

1 Simple Instruction Substitution 2
2 Mistake in Control Flow for Branch Delay Slot 4
3 Unconditional Branch Instrumentation 5
4 Conditional Branch Instrumentation 6
5 Instrumented Code with 'Stepping Stone' to Correct Long Branch Errors . 9
6 Operand Length Histogram 10
7 Data Cache Miss Rate 11

1 What Is RYO

RYO (Roll Your Own) is a family of novel instrumentation tools for the PA-RISC family of processors [3]. These tools replace a specific set of PA-RISC assembly instructions with user supplied subroutines.

Because the user provides his own custom instrumentation routines, the use of the tool is virtually unlimited. Its uses could range from replacing a faulty hardware divide instruction with a correct software substitute, to generating address trace files for later analysis. Like Pixie[5], RYO generates program analysis output by running an instrumented executable, yet RYO benefits from both a simpler implementation and greater flexibility by its use of arbitrary user supplied subroutines.

This paper introduces the overall structure and design of RYO, as well as giving detailed instructions on its use. A specific member of the RYO family, RYOLS (Roll Your Own Load Store), is presented to illustrate correct program usage. The final section shows actual experimental data obtained with two members of the RYO family, RYOLS and RYOFP (Roll Your Own Floating Point).

1.1 Overview

RYOLS expects an assembly level, or .s, file for input. It makes two passes through the program and replaces memory operations (floating point operations in the case of RYOFP) with an unconditional jump to the RYOLS library calling stub. This stub saves the state of the machine, does any necessary parameter passing, calls the proper RYOLS library procedure, and finally unconditionally jumps back to the original execution sequence (figure 1).

The overall design simplicity is an attractive feature, yet actual implementation involves overcoming a number of pesky problems. For this purpose, a number of small companion programs–fixtargets, offset.awk, and lbranch.awk–have also been implemented.

1.2 Advantages and Disadvantages

Using this relatively simple substitution strategy has a number of advantages and disadvantages. On the plus side, substituting a single branch instruction for the single memory instruction results in an exact one for one instruction substitution that leaves the relative positions of the instructions unchanged. Of course, code must be inserted to build the library calling stubs, but this code is placed between procedures where it can do little harm. Keeping the relative instruction position constant not only eliminates the need to translate a branch to a target plus a constant offset to a corrected offset, but is also the only way to ensure correct indirect branching. In the case of indirect branching, the target address cannot be known until runtime, so the offset cannot be translated when doing the instrumentation.

Another advantage is the use of high level C-language routines for instruction substitution. This makes it relatively simple to implement arbitrarily complex on-the-fly analysis. Finally, because the application code is actually executed along with the extra analysis routines, verification is performed simply by comparing program output before and after instrumentation. For example, when instrumenting an mpeg decoder, many subtle bugs

original code:

```
           ...
        .SPACE  $TEXT$,SORT=8
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=0x2c,CODE_ONLY,SORT=24
DoDitherImage
        .PROC
        .CALLINFO CALLER,FRAME=16,SAVE_RP,ARGS_SAVED
        .ENTRY
        STW     %r2,-20(%r30)
        LDO     64(%r30),%r30
        ADDIL   LR'ditherType-$global$,%r27
           ...
```

modified code:

```
           ...
        .SPACE  $TEXT$,SORT=8
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=0x2c,CODE_ONLY,SORT=24
DoDitherImage
        .PROC
        .CALLINFO CALLER,FRAME=16,SAVE_RP,ARGS_SAVED
        .ENTRY
$n1     b,n $m1  ;         STW     %r2,-20(%r30)    ; substituted instruction
        LDO     64(%r30),%r30
        ADDIL   LR'ditherType-$global$,%r27
           ...
```

procedure calling stub:

```
$m1
        stw %r2,156(0,%r30)                         ; save machine state
        bl $saveall,%r2
        nop
        copy %r30,%r25 ;copy basereg to %r25        ; pass parameters
        bl .+12,%r26
        ldo -3(%r26),%r26
        STW     %r2,-20(%r30)                       ; copy op to pass to ryo_word
        bl      ryo_word,%r2                        ; call instrumentation
        ldo 320(%r30),%r30                          ;        routine
        bl $restall,%r2                             ; restore machine state
        ldo -320(%r30),%r30
        STW     %r2,-20(%r30)                       ; do mem op
        b,n $n1+4                                   ; return to normal
                                                    ;        program flow
```

Figure 1: Simple Instruction Substitution

were noticed only because the video image displayed did not look quite the way it did when running the uninstrumented code.

The major disadvantage of this tool is that source code is required to do the instrumentation. This relies on the compiler to correctly generate an intermediate assembly output which does not always happen. Furthermore, library calls and other pieces of code dynamically linked in to the executable will not be instrumented. Finally, in the current implementation, no information is maintained concerning basic blocks and cycles between instrumented instructions. This strategy was chosen to make implementation simpler in that the the high level instrumentation routine does not know any global information about the program state. It only knows that a given instruction was executed. These drawbacks could be corrected in future revisions if necessary.

1.3 Implementation

In order to cause a one for one instruction substitution, the basic strategy is to replace a targeted instruction with an unconditional jump to a stub, save the machine state, and unconditionally jump back to the instruction following the targeted instruction (see figure 1). If the return address was not hard coded at the time of instrumentation, it would be necessary to store a return address in a register, and thereby alter the machine state.

1.3.1 Branch Delay Slot

This works correctly for the most part, except that a problem arises when a load or store instruction falls in the branch delay slot of a preceding branch instruction. In this case, a branch would be substituted in the branch delay slot. The instruction instrumentation routine will be correctly called as anticipated, but the final branch will be to the return point statically calculated at instrumentation time. Thus, the control flow will return to the second instruction following the branch rather than the correct branch target (figure 2). Note that the branch placed in the branch delay slot of the initial branch correctly modifies control flow when a nullifying branch is used.

This problem is corrected as shown in figure 3. Now the branch instruction with a load or store following it is also targeted for replacement. It now has its own stub with an appropriate instrumentation call. This time, though, the final instruction in the stub is not the load or store instruction as before, but instead the branch with the load or store in the delay slot. The final branch returns to the second instruction past the branch, since the instruction immediately following was already executed in the delay slot. Furthermore, the initial instruction in the stub is the same branch followed by another branch in the delay slot that branches to the next sequential instruction. This is so to determine if the instruction in the delay slot would have been nullified. If it was nullified, then the branch to the next sequential instruction will now be nullified instead, and the branch will go to the intended target. If the instruction was not nullified, then the branch in the delay slot will be executed, and the control flow will continue sequentially. Additionally, the load or store instruction in the main program body is replaced with an unconditional branch to its own stub as before.

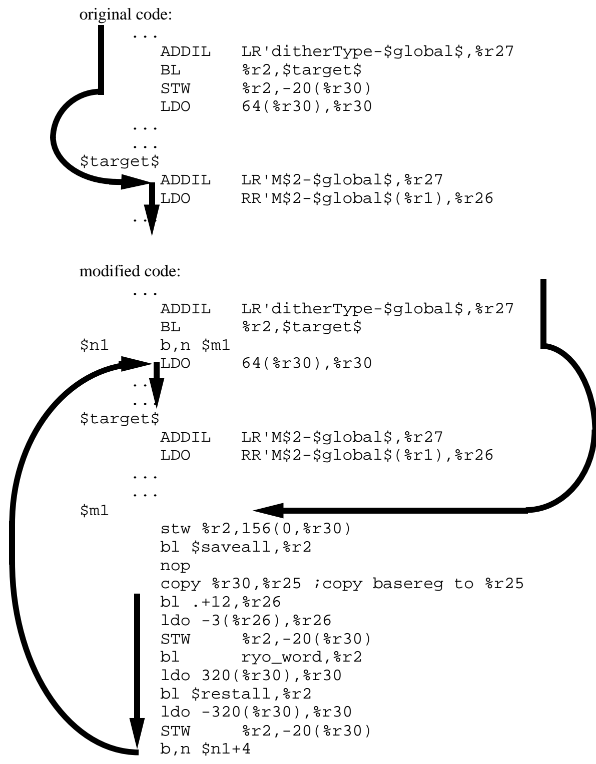

Figure 2: Mistake in Control Flow for Branch Delay Slot

initial code:

```
        ...
$L0
        B       $00000005
        STW     %r29,-52(%r30)
        ...
```

modified code:

```
        ...
$L0
$n5     b,n $m5                                 ; B      $00000005
$n6     b,n $m6                                 ; STW %r29,-52(%r30)
        ...
```

procedure calling stubs:

```
$m5
        B       $00000005
        b,n     .+4                             ; test for branch nullification
        stw %r2,156(0,%r30)
        bl $saveall,%r2
        nop
        copy %r30,%r25
        bl .+12,%r26
        ldo -3(%r26),%r26
        STW     %r29,-52(%r30)
        bl      ryo_word,%r2
        ldo 320(%r30),%r30
        bl $restall,%r2
        ldo -320(%r30),%r30
        B       $00000005                       ; execute branch and instruction
        STW     %r29,-52(%r30)                  ;       in branch delay slot
        b,n     $n5+8
$m6
        ...
        ...                                     ; simple substitution routine as in fig 1
        ...
        b,n     $n6+4
```

Figure 3: Unconditional Branch Instrumentation

At this point one might wonder why it is necessary to replace both the branch and the instruction in the delay slot with branches to stubs. Since the instrumentation routine for the load or store in the delay slot is called from the stub for the branch, why is it necessary for the instruction in the delay slot to have its own calling stub? This is because it is possible for the instruction in the delay slot to be a target for some previous jump. In this case it should be executed as a standalone instruction and not in the delay slot of a branch.

A final complication is added in the case of an unconditional branch. Recall that the first instructions in the calling stub are the original branch and a branch in its delay slot (see figure 3). This was done to determine if the delay slot instruction should be nullified. For unconditional branches, the nullification is dependent upon the direction of the branch, so that care must be taken to ensure that the branch direction is the same as it was in the original code. This is done by having the branch go to a forward or reverse stub, which then branches to the original target (see figure 4). The decision whether to use a forward or reverse stub can be determined at instrumentation time by comparing the address of the branch instruction with the target. In this final form, all branches will be correctly instrumented and executed.

initial code:

```
            ...
        ADDIL   LR'ditherType-$global$,%r27
        COMIB,=,N       0,%r8,$0002004A ;offset 0x2ac
        LDWX,S  %r9(%r7),%r26
            ...
```

modified code:

```
            ...
        ADDIL   LR'ditherType-$global$,%r27
$n115   b,n $m115                                   ; COMIB,=,N  0,%r8,$0002004A
$n116   b,n $m116                                   ; LDWX,S  %r9(%r7),%r26
            ...
```

procedure calling stub:

```
$m115
        COMIB,=,N       0,%r8,$forwstub115          ; replace target field
        b,n     .+4
        stw %r2,156(0,%r30)
        bl $saveall,%r2
        nop
        copy %r9,%r24
        copy %r7,%r25
        bl .+12,%r26
        ldo -3(%r26),%r26
        LDWX,S  %r9(%r7),%r26
        bl      ryo_wx,%r2
        ldo 320(%r30),%r30
        bl $restall,%r2
        ldo -320(%r30),%r30
        COMIB,=,N       0,%r8,$forwstub115          ; do branch and op in delay slot
        LDWX,S  %r9(%r7),%r26
        b,n     $n115+8
$forwstub115
        b,n     $0002004A
```

Figure 4: Conditional Branch Instrumentation

1.3.2 Saving and Restoring Machine State and Parameter Passing

After it has been determined that the load or store instruction targeted is not to be nullified, the state of the machine must be stored. This is done simply by copying the register set

to the stack. All caller-save general purpose and floating point registers, as well as relevant special registers such as the shift amount register, and space id registers are copied. The stack pointer is then incremented to point to the new top of stack. Performance could be improved by saving only those registers currently in use, but doing the necessary dependency checking would probably greatly complicate the implementation.

Next, the appropriate parameters must be copied to the correct registers for passing to the instrumentation routine. First, a BL instruction is executed such that the return address points to the targeted load store instruction. This pointer is passed to the instrumentation routine so that the instruction in machine format is available for analysis. In most cases, the base address and offset are also passed as parameters, but this can vary for different types of memory operations.

After the instrumentation routine is called, the machine state is restored in the opposite way from which it was saved, and the control flow is returned to the next proper instruction. In this way, when the program is executed, all instructions will be executed as normal, but instrumentation routines will also be called for all targeted instructions.

2 How to Use RYO

RYOLS is really an awk script that takes an assembly file as input. An instrumented .s file is output and assembled to an object file. The object file is then linked with the instrumentation library to create an instrumented executable. If one were to instrument the C program test.c, for example, one would type :

```
cc -S -O test.c
awk -f ryols.awk <test.s >test.fs
as -o test.o test.fs
cc test.o lib.c -o test_inst.
```

2.1 Compiler Errors

This assumes that the .s assembly file is correct. Unfortunately, the HP C compiler does not always produce a correct .s file. The authors have found two problems for which solutions are given below, however, it is likely that other compiler bugs exist. Therefore, if the instrumented code does not behave as expected, the first thing to check is to see if it runs correctly without instrumentation, but compiled through an intermediate assembly language step. For the above example, one would see if the executable created with :

```
cc -S -O test.c
as -o test.o test.s
cc test.o -o test_s
```

behaves the same as one compiled in the normal way. The GNU C compiler actually uses an intermediate .s file in program compilation, since it generates object code with the native assembler. For this reason, one would expect generation of a correct .s file when compiling with GCC, however, use of the non-native compiler may generate less efficient code.

2.1.1 Branch Targets

Occasionally, the compiler will lose track of branch targets and will substitute "???" for the branch target field. If left uncorrected, this will cause an error in the assembler. The tool fixtargets fixes this problem.

Fixtargets requires the existence of a correct .o and the flawed .s file. These can be created with :

```
cc -S -c test.c.
```

Next, type :

```
fixtargets test
```

and the correct .o code will be disassembled to provide the correct branch targets for substitution into the .s file. Branches to a symbolic target plus an offset are handled such that the offset is always in decimal.

2.1.2 Indirect procedure calls

If a procedure name is used as a parameter passed to another procedure, incomplete assembly code is generated. If, for example, a procedure named key is the parameter to another procedure call, the compiler will generate :

```
        .WORD key.
```

towards the end of the .s file. This must be altered so that it instead reads :

```
        .WORD  P'key.
```

2.2 Long Branches

In some cases, so much instrumentation code will be added to the executable that the branch target field will not have enough bits to hold the correct target. This usually occurs in branches to procedures in other modules. In this case, the linker will try to build a procedure stub at the beginning of the object file. If the object file is very large, then branches from the end of the file will not be able to reach the beginning of the file. This is usually seen during link time with an error like :

```
The value 0xfffbffcc did not fit into a 19 bit field at offset 0x4024c
```

In this case one should execute these instructions :

```
cp test.fs test.fs.bad
awk -f offset.awk <test.fs.bad | awk -v StartFix=262632 -f lbranch.awk >test.fs
```

original code:

```
            ...
        copy %r30,%r25
        .CALL   ARGW0=GR,ARGW1=GR,RTNVAL=GR
        BL  printf,%r2
        b,n  .+4
            ...
```

modified code:

```
            ...
printf_stub
        .CALL   ARGW0=GR,ARGW1=GR,RTNVAL=GR
        bl,n  printf,%r0
            ...
            ...
            ...
        copy %r30,%r25
        BL  printf_stub,%r2
        b,n  .+4
            ...
```

Figure 5: Instrumented Code with 'Stepping Stone' to Correct Long Branch Errors

In this case, 262632 is the offset given in the above error message (0x4024c) converted to decimal less 100. The factor of 100 is an error margin to allow for insertion of extra instructions and can be made larger or smaller at the user's discretion.

This will create a 'stepping stone' for all procedure calls after the offset where the error was encountered. A 'stepping stone' is an intermediate branch target followed by a branch to the original target so that the long branch is broken into two shorter ones (figure 5).

Note that this strategy will only work up to a certain point. The authors have not encountered any cases where it did not work, but this does not guarantee that it always will. Some manual tweaking of the branch targets may be called for. Furthermore, if this does not work the first time and further iterations of lbranch.awk are required, one must always resume from the original file, test.fs.bad.

2.3 Offset.awk

Finally, offset.awk is a utility provided to calculate the new instruction offsets after the instrumentation code has been added. To use offset.awk, simply type :

```
awk -f offset.awk <test.fs >test.fs.off
```

Test.fs.off can be modified and run through multiple iterations of offset.awk.

3 Example Data

This section presents experimental data actually obtained with RYOLS and RYOFP. Two examples of data are presented to illustrate how varying the instructions targeted for replacement and the instrumentation library can produce totally different types of data. The first example shows arithmetic data generated from a JPEG [6] decoder, and the second shows address traces generated from mpeg_play [4].

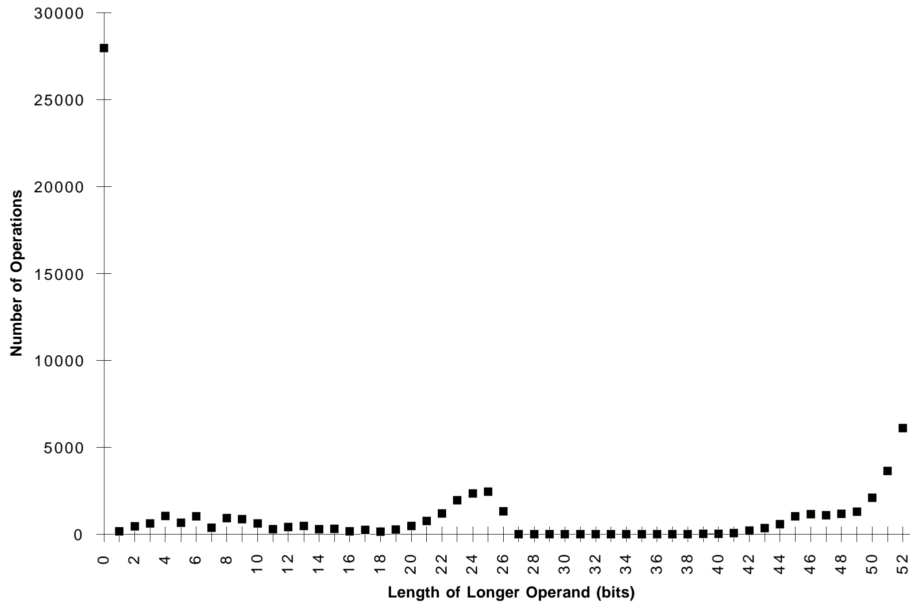

Figure 6: Operand Length Histogram

3.1 Arithmetic

To generate this data, RYOFP was run on djpeg, the Independent JPEG Group's JPEG software decoder modified to use floating point for the DCT calculations. It was also modified to use DE arithmetic enhancement for increased performance [7]. Floating point operations were selected for instrumentation. The instrumentation library dynamically calculated the length of operands in bits for each floating point arithmetic operation encountered when executing the instrumented code on the image Lena, and wrote the longer of the two operands to an output file. Finally, a graph was constructed to show a histogram of operand lengths.

The most interesting thing to note from the data is the large number of zero length operands. A compressed image will tend to have many zero elements in each 8x8 macro block, and this is clearly indicated in the experimental data obtained.

There is also a noticeable hump at approximately 27 bits. This is an effect of DE calculation methodology. In this case, operands are 27 bits long when one of the two operands packed into a single double precision data word equals zero.

3.2 Memory

This data was obtained by using RYOLS to instrument load and store instructions in mpeg_play, Berkeley's MPEG [1] player. The instrumentation library simply printed the type of operation and the instructions address to an output trace file. In this case, not only load and store instructions, but also the library routine fread was instrumented. It was observed that all input data was coming through fread, so this routine was additionally

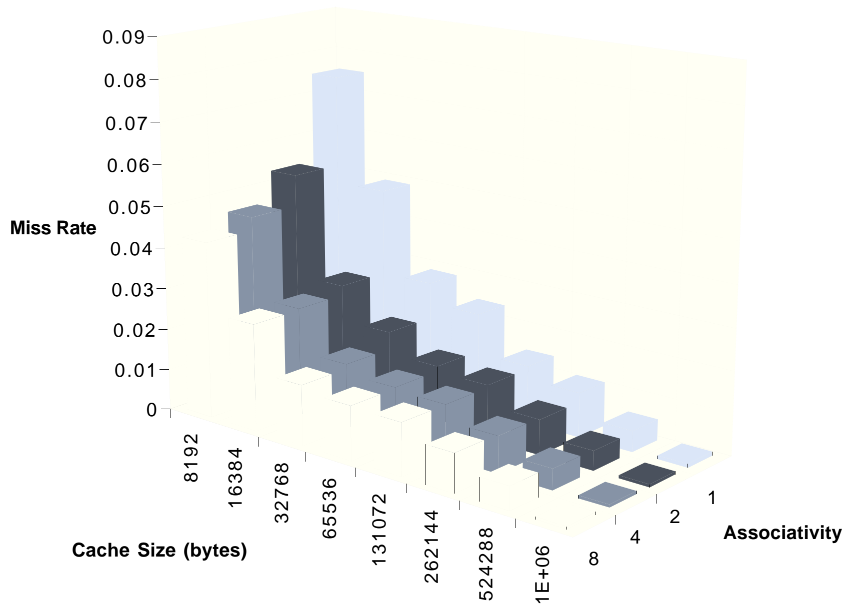

Figure 7: Data Cache Miss Rate

instrumented to simulate more accurate memory behavior.

These address traces were then used as an input to dineroIII [2], a popular cache simulator. The same address trace file was used to generate cache miss rates for many different cache designs. The data is shown in figure 7. It is interesting to note that for the movie played, waterski.mpg, virtually all the data references are captured with a cache of size 1MB. Finally for smaller sized caches, changing from direct mapped to 2 way gives approximately the same effect as doubling the cache size.

4 Conclusion

RYOLS and RYOFP are two general purpose tools from the RYO family of instumentation tools for the PA-RISC architecture. Since an arbitrary instrumentation routine is substituted for designated instructions, the utility of the tool is limited only by the creativity of the user. For information on obtaining RYOLS email the authors at zucker@umunhum.stanford.edu or karp@hpl.hp.com.

5 Acknowledgments

The authors would like to thank Michael J. Flynn and Ruby B. Lee for their overall support of this work, and brians@cup.hp.com for his help with the HP C compiler.

A Source Code for Ryols.awk

```
#
# This awk script takes a .s file as input and produces a .s file with
# all memory ops replaced by subroutine calls.  It will
# handle nullified ops correctly.
# If you use any substantive part of this work, please acknowledge the
# source.
#
# I'll try to help, but I make no promises.
#
# Alan H. Karp, HP Labs, karp@hpl.hp.com
# Last modified June 30, 1993
#
# Modified by Dan Zucker, Stanford University, January, 1994
# Modified again September, 1994 to generate address traces
# Modified January, 1995
# zucker@umunhum.stanford.edu
#
# special command line options:
#    -v proc=myprocname       instruments only the named procedure
#
#
# Initialize
#
BEGIN {

    offcount = -4
    labelcount = 0
    label = 0
    nh = 0
    target = 0
    arch = "DA1.0" # DA1.1 for 32 fprs and 5-ops
    for ( i = 1; i <= 200; i++ ) hist[i] = 0
#
# Routine to call for each op_type
#
   nroutines = 0
   routine[nroutines++] = "ryo_error"
   routine[nroutines++] = "ryo_word"
   routine[nroutines++] = "ryo_halfword"
   routine[nroutines++] = "ryo_byte"
   routine[nroutines++] = "ryo_wm"
   routine[nroutines++] = "ryo_wx"
```

```
   routine[nroutines++] = "ryo_hx"
   routine[nroutines++] = "ryo_bx"
   routine[nroutines++] = "ryo_wax"
   routine[nroutines++] = "ryo_cwx"
   routine[nroutines++] = "ryo_ws"
   routine[nroutines++] = "ryo_hs"
   routine[nroutines++] = "ryo_bs"
   routine[nroutines++] = "ryo_was"
   routine[nroutines++] = "ryo_cws"
   routine[nroutines++] = "ryo_bys"
   routine[nroutines++] = "ryo_fwx"
   routine[nroutines++] = "ryo_fdx"
   routine[nroutines++] = "ryo_fws"
   routine[nroutines++] = "ryo_fds"
   routine[nroutines++] = "ryo_fread"
#
# Define caller saves registers
#
    grs = "%r1 %r2 %r19 %r20 %r21 %r22 %r23 %r24 %r25 %r26 %r27 %r28 %r29 %r31 "
    frs = "%fr4 %fr5 %fr6 %fr7 %fr8 %fr9 %fr10 %fr11 "
    if ( arch == "DA1.1" )
       frs = frs "%fr22 %fr23 %fr24 %fr25 %fr26 %fr27 %fr28 %fr29 %fr30 %fr31 "   # Used
    srs = "%sr0 %sr1 %sr2 %sr3"  # Is sr3 needed?
    crs = "%cr11"
    caller_saves = grs " " frs " " srs " " crs
    ncs = split(caller_saves,csr)
    frs_calle = "%fr12 %fr13 %fr14 %fr15 %fr16 %fr17 %fr18 %fr19 %fr20 %fr21 "
    nce = split(frs_calle,cse)
#
#
# Generate instructions to save and restore registers
#
    k = 0
    n = split(frs,r)
    for ( i = 1; i <= 2; i++ ) {
        save[r[i]] = " fstds " r[i] "," k "(0,%r30)"
        rest[r[i]] = " fldds " k "(0,%r30)," r[i]
        k = k + 8
    }
    for ( i = 3; i <= n; i++ ) {
        save[r[i]] = " ldo "  k "(%r30),%r1" ORS " fstdx " r[i] ",0(0,%r1)"
        rest[r[i]] = " ldo "  k "(%r30),%r1" ORS " flddx 0(0,%r1)," r[i]
        k = k + 8
    }
```

```
    n = split(frs_calle,r)
    for ( i = 1; i<= n; i++ ) {
        save[r[i]] = " ldo "  k "(%r30),%r1" ORS " fstdx " r[i] ",0(0,%r1)"
        rest[r[i]] = " ldo "  k "(%r30),%r1" ORS " flddx 0(0,%r1)," r[i]
        k = k + 8
    }
    save["%fr0"] = " ldo "  k "(%r30),%r1" ORS " fstdx " r[i] ",0(0,%r1)"
    rest["%fr0"] = " ldo "  k "(%r30),%r1" ORS " flddx 0(0,%r1)," r[i]
    k = k + 8
    n = split(grs,r)
    for ( i = 1; i <= n; i++ ) {
        save[r[i]] = " stw " r[i] "," k "(0,%r30)"
        rest[r[i]] = " ldw " k "(0,%r30)," r[i]
        k = k + 4
    }
    n = split(srs,r)
    for ( i = 1; i <= n; i++ ) {
        save[r[i]] = " mfsp " r[i] ",%r1" ORS " stw %r1," k "(0,%r30)"
        rest[r[i]] = " ldw " k "(0,%r30),%r1" ORS " mtsp %r1," r[i]
        k = k + 4
    }
    n = split(crs,r)
    for ( i = 1; i <= n; i++ ) {
        save[r[i]] = " mfctl " r[i] ",%r1" ORS " stw %r1," k "(0,%r30)"
        rest[r[i]] = " ldw " k "(0,%r30),%r1" ORS " mtctl %r1," r[i]
        k = k + 4
    }
#
# Calculate additonal stack size needed
# Round up to 64 byte boundary
#
    added_stack = 64 + 64*int((k+63)/64)
    stack_save = " ldo "  added_stack "(%r30),%r30"
    stack_rest = " ldo -" added_stack "(%r30),%r30"
#
# Put out code sequences to save and restore all registers
#
#
# Probably don't need to save frs_calle but better safe...
#
save_restore_reg = caller_saves #"  " frs_calle
nsr = split(save_restore_reg, srreg)

    # add saveall,restall code
```

```
    hold[++nh] = "$saveall"
    for ( j = 1; j <= nsr; j++ )
      if ( srreg[j] != "%r2" ) hold[++nh] = save[srreg[j]]
    hold[++nh] = rest["%r1"]            #added 10-21-94
    hold[++nh] = " bv 0(%r2)"
    hold[++nh] = rest["%r2"]
#
# Set up to restore all registers before a branch
#
    hold[++nh] = "$restall"
    for ( j = nsr; j > 0; j-- )
      if ( srreg[j] != "%r2" ) hold[++nh] = rest[srreg[j]]
    hold[++nh] = " bv 0(%r2)"
    hold[++nh] = rest["%r2"]
#
# Set up to save all registers following a branch target
}
#
# Read file and keep track of branch targets and op types
#
{
 line[NR] = $0
 split($0,op," ")
#
# Determine instruction type
#
 op_type[NR] = "int"
 if ( substr(op[1],1,1) == "." ) op_type[NR] = "directive"
 if ( index(op[1],",SGL") != 0 ) op_type[NR] = "float"
 if (substr(op[1],1,3) == "FLD") op_type[NR] = "load"
 if (substr(op[1],1,3) == "FST") op_type[NR] = "store"
 if ( index(op[1],",DBL") != 0 ) op_type[NR] = "double"
 if (substr(op[1],1,2) == "LD" && substr(op[1],3,1) != "I" &&
    op[1] != "LDO" && op[1] != "LDSID") op_type[NR] = "load"
 if (substr(op[1],1,2) == "ST" ) op_type[NR] = "store"
 split(op[1],opc,",")
 if ( opc[1] == "MOVB"   ||
      opc[1] == "MOVIB"  || opc[1] == "COMB"   ||
      opc[1] == "COMBT"  || opc[1] == "COMBF"  ||
      opc[1] == "COMIBT" || opc[1] == "COMIBF" ||
      opc[1] == "COMIB"  || opc[1] == "ADDB"   ||
      opc[1] == "ADDBT"  || opc[1] == "ADDBF"  ||
      opc[1] == "ADDIBT" || opc[1] == "ADDIBF" ||
      opc[1] == "ADDIB"  ||
```

```
      opc[1] == "BVB"    || opc[1] == "BB"     ||
      opc[1] == "COMIB"  )
    op_type[NR] = "cbranch"
 if ( opc[1] == "BL"     || opc[1] == "BLR"    ||
      opc[1] == "BV"     || opc[1] == "BE"     ||
      opc[1] == "BLE"    || opc[1] == "B"      )
    op_type[NR]= "branch"
 if ( substr($1,1,1) == ";" ) op_type[NR] = "comment"
 if ( substr($0,1,1) != " " && substr($0,1,1) != "\t") op_type[NR] = "target"
 if ( $0=="") op_type[NR] = "nothing"

# hacks -- causes intentional syntax errors
if (substr(op[1],1,4) == "GATE") print "GATE instruction found"

#
# Compute offsets
#
 if ( op_type[NR] != "target" && op_type[NR] != "comment" &&
      op_type[NR] != "directive" && op_type[NR] != "nothing") {
#      length($0) > 1 )
    offcount = offcount + 4
    offset[NR] = offcount
;    ilinenum[offcount] = NR
 } else offset[NR] = offcount + 4

# keep track of target offset
 if (op_type[NR] == "target") toffset[op[1]] = offset[NR]

}

function out_ryo_code(lineout) {
      nopc=split(lineout,opc)
                                            type =  0 # Bad op code
      if ( substr(opc[1],3) == "W"  ) type =  1 # word
      if ( substr(opc[1],3) == "H" ) type =  2 # halfword
      if ( substr(opc[1],3) == "B"  ) type =  3 # byte
      if ( substr(opc[1],3) == "WM" ) type =  4 # word and modify
      if ( substr(opc[1],3,2) == "WX" ) type =  5 # word indexed
      if ( substr(opc[1],3,2) == "HX" ) type =  6 # halfword indexed
      if ( substr(opc[1],3,2) == "BX" ) type =  7 # byte indexed
      if ( substr(opc[1],3,3) == "WAX" ) type =  8 # word absolute indexed
      if ( substr(opc[1],3,3) == "CWX" ) type =  9 #and clear word indexed
      if ( substr(opc[1],3,2) == "WS" ) type =  10 # word short
      if ( substr(opc[1],3,2) == "HS" ) type =  11 # halfword short
```

```
      if ( substr(opc[1],3,2) == "BS" ) type =  12 # byte short
      if ( substr(opc[1],3,3) == "WAS" ) type =  13 # absolute short
      if ( substr(opc[1],3,3) == "CWS" ) type =  14 # and clear word short
      if ( substr(opc[1],3,3) == "BYS" ) type =  15 # bytes short
      if ( substr(opc[1],4,2) == "WX" ) type =  16 # float word indexed
      if ( substr(opc[1],4,2) == "DX" ) type =  17 # float double indexed
      if ( substr(opc[1],4,2) == "WS" ) type =  18 # float word short
      if ( substr(opc[1],4,2) == "DS" ) type =  19 # float double short
      if ( substr(opc[1],1,2) == "BL" &&
           substr(opc[2],1,5) == "fread") type =  20 # fread

#
#  save registers
#
        hold[++nh] = save["%r2"]
        hold[++nh] = " bl $saveall,%r2"
        hold[++nh] = " nop"
#        hold[++nh] = rest["%r1"]
#
# get pointer to op
#
#       hold[++nh] = stack_save

        nreg = split(opc[2],reg,"[()]")
        nsubspace = split(reg[2],subspace,",")
        if (nsubspace == 1) basereg = subspace[1]
          else basereg = subspace[2]
if (type != 20) {
        if (type>=5 && type <= 9) {
          indexreg = reg[1]
          hold[++nh] = " \tcopy " indexreg ",%r24  ;copy indexreg to %r24"
        }
        if ((type>=5 && type <= 9) && basereg=="%r24") {
          temp = rest["%r24"]
          sub("%r24","%r25",temp)
          hold[++nh] = temp "; restore basereg from memory"
        }
else hold[++nh] = " \tcopy " basereg ",%r25 ;copy basereg to %r25"

        hold[++nh] = " \tbl .+12,%r26"
        hold[++nh] = " \tldo -3(%r26),%r26"
        hold[++nh] = lineout
}
```

```
#
# Set up for call
#



#increment stack pointer and call correct routine

        hold[++nh] = " bl\t" routine[type] ",%r2"
        hold[++nh] = stack_save
#        hold[++nh] = "        nop"

#
# restore register set
#
        hold[++nh] = " bl $restall,%r2"
        hold[++nh] = stack_rest
#           hold[++nh] = " nop"
#           hold[++nh] = rest["%r2"] # is this right?
}

#
# Now process file
#
END {


#
# Replace load/stores with branches
#
# second iteration through instruction stream
  for ( i = 1; i <= NR; i++ ) {
#
# Parse current line
#
    split(line[i],op," ")
    n = split(op[1],opc,",")
    nreg = split(op[2],r,"[(),]")

    if ( op[1] == ".END" ) {
      for ( j = 0; j < nroutines; j++ )
        print "\t.import " routine[j] ",code\t\t; Added ryofp_s"
    }
```

```
    if (op_type[i+1] == "directive") {
      na = split(line[i+1],a)
      if (a[1]==".PROC") {
        if (op_type[i] != "target") print "error of type 1"
        else dothisprocedure = (proc == op[1])
      }
    }
    na = split(line[i-1],a)
    if (a[1]==".PROCEND") dothisprocedure=0


    if (dothisprocedure==1 || proc=="") {
       #only instrument certain procedures
#
# found a load/store
#
      if ( op_type[i]=="load" || op_type[i]=="store") {
#
# Replace op with branch and build code to make call
#
      label++ # Label counter
      print "$n" label "\tb,n $m" label "  ; " line[i] "  " op_type[i]

#
# Code to build stub routine begins here
#

      hold[++nh] = "$m" label
      out_ryo_code(line[i])

      hold[++nh] = line[i]
      hold[++nh] = "\tb,n $n" label "+4"
#end build code stub

    } else if (op_type[i] == "cbranch") {
      origline = line[i]

      #find next real instruction
      nextline = i+1
      while((op_type[nextline]=="target" || op_type[nextline]=="directive" ||
             op_type[nextline]=="comment" || op_type[nextline]=="nothing") &&
             nextline<=NR) nextline++
      if (op_type[nextline] =="load" || op_type[nextline] =="store") {
        label++
```

```
print "$n" label "\tb,n $m" label " ; "line[i]

#determine target and target direction
nl = split(line[i],op," ")
nm = split(op[2],field,",")
target_field = field[nm]

targ_offset = toffset[target_field]
if (index(target_field,"+") != 0 ) {
  no = split(target_field,operand,"+")
  if (operand[1]==".") operand[1] = offset[i]
  targ_offset = toffset[operand[1]] + operand[2]
}

if ( index(target_field,"-") != 0) {
  no = split(target_field,operand,"-")
  if (operand[1]==".") operand[1] = offset[i]
  targ_offset = toffset[operand[1]] - operand[2]
}

if (offset[i]-targ_offset >0) forwbranch=0
   else forwbranch = 1

if (forwbranch) {
  new_targ = "$forwstub" label
} else {
  hold[++nh] = "$backstub" label
  hold[++nh] = "\tb,n \t" target_field
  new_targ = "$backstub" label
}

# substitute new target field
op[2] = ""
for (k=1; k<=nm-1; k++)
  op[2] = op[2] field[k]","
op[2]=op[2] new_targ
line[i] = "\t" op[1] "\t" op[2]
for (k=3; k<=nl; k++)
  line[i]= line[i] "  " op[k]
line [i] = line[i]" ; -- "origline

hold[++nh] = "$m" label

hold[++nh] = line[i]
```

```
        hold[++nh] = "\tb,n \t.+4"

        out_ryo_code(line[nextline]) #bug with type for the moment
#
# fix addib immediate field
#
        # change immediate value to zero in ADDIB
        if (substr(op[1],1,5) == "ADDIB") {
          numop=split(line[i], op, " ")
          numop2 = split(op[2],op2,",")
          tempa = "\t" op[1] "\t0"
          for (k = 2; k <= numop2; k++)
            tempa = tempa "," op2[k]
          for (k = 3; k<= numop; k++)
    tempa = tempa "   " op[k]
          line[i] = tempa
        }

        hold[++nh] = line[i]
        hold[++nh] = line[nextline]
        hold[++nh] = "\tb,n \t$n" label "+8"

        if (forwbranch == 1) {
          hold[++nh] = "$forwstub" label
          hold[++nh] = "\tb,n \t" target_field
        }
      } else print line[i]
    }
    else if ( op[1] == ".CALL" && op_type[i+1] == "branch") {
      #found a procedure call
      #find next real instruction
      nextline = i+2
      while((op_type[nextline]=="target" ||
             op_type[nextline]=="directive" ||
             op_type[nextline]=="comment" ||
             op_type[nextline]=="nothing") &&
             nextline<=NR) nextline++
      numa = split(line[i+1],a," ")
      numb = split(a[2],b,",")

      # look for specific library calls
      if (b[1]=="fread") {
        label++
        print "$n" label "\tb,n $m" label " ; special library call "line[i+1]
```

```
    hold[++nh] = "$m" label

    # substitue destination
    tempout = "\t" a[1] "\t$nullinstr" label ",%r0"
    for (j=3; j<=numa; j++) tempout = tempout "\t" a[j]

    hold[++nh] = tempout   #BL or BLE
    hold[++nh] = "\tb,n \t.+4"

    if (op_type[nextline] =="load" || op_type[nextline] =="store") {
    # fread with load/store in delay slot
      out_ryo_code(line[nextline])
      hold[++nh] = line[nextline]
    } else {
    # fread with normal op in delay slot
      hold[++nh] = line[nextline]
    }

    hold[++nh] = "$nullinstr" label
    out_ryo_code(line[i+1])
    hold[++nh] = line[i]
    hold[++nh] = line[i+1]
    hold[++nh] = "\tnop"
    hold[++nh] = "\tb,n \t$n" label "+8"
    i++
  } else {
    if (op_type[nextline] =="load" || op_type[nextline] =="store") {
      label++
      print "$n" label "\tb,n $m" label " ; .CALL "line[i+1]

      hold[++nh] = "$m" label
      hold[++nh] = line[i]     #.call
      hold[++nh] = line[i+1]   #BL or BLE
      hold[++nh] = "\tb,n \t.+4"
      out_ryo_code(line[nextline])
      hold[++nh] = line[i]     #.call
      hold[++nh] = line[i+1]
      hold[++nh] = line[nextline]
      hold[++nh] = "\tb,n \t$n" label "+8"
      i++;
    } else print line[i]
  }  # op type branch but not a specific library call
} else if (op_type[i] == "branch") {
  #not procedure call
```

```
      #find next real instruction
      nextline = i+1
      while((op_type[nextline]=="target" ||
             op_type[nextline]=="directive" ||
             op_type[nextline]=="comment" ||
             op_type[nextline]=="nothing") &&
             nextline<=NR) nextline++
      if (op_type[nextline] =="load" || op_type[nextline] =="store") {

        label++
        print "$n" label "\tb,n $m" label " ; "line[i]
        hold[++nh] = "$m" label
        hold[++nh] = line[i]
        hold[++nh] = "\tb,n \t.+4"
        out_ryo_code(line[nextline])
        hold[++nh] = line[i]
        hold[++nh] = line[nextline]
        hold[++nh] = "\tb,n \t$n" label "+8"
      } else print line[i]
    } else {
      #not a branch,load, or store

        if ( op[1] == ".PROCEND" ) {
           print "; ------ Start code added by ryofp ---------------"
#
# Print hold[nh] instructions for each routine instructions
#
           for ( j = 1; j <= nh; j++ ) print hold[j]
           nh = 0
           print "; ------ End code added by ryofp -----------------"
        }

        #output instruction
        print line[i]
      } #end of else--not a load or store
      } else { # code that's not being instrumented
        print line[i]
      }
# Set up for next line
#
#      if ( op_type[i] != "target" && op_type[i] != "directive" &&
#          op_type[i] != "comment" ) prev_inst_no = i

  } #for i loop
```

```
} #END
```

B An Example Instrumentation Routine

```
basic_instr(pointer,basereg)
int *pointer,basereg;
{
  int i,j,k;

  /* get immediate field */
  j = *pointer;
  j = j & 0x3fff;  /* right most 14 bits */
  k = 1<<13;

  if (j&1) /* sign extend */ {
    j = j | 0xffffc000;  /* left most 18 bits */
  }

  j = j>>1; /* get rid of sign bit (arithmetic shift)*/

  fprintf(fpryo,"%x", basereg+j);
  fprintf(fpryo,"\t%x",pointer);

#if verbose
  fprintf(fpryo,"\tLD/ST W/H/B \n");
  fprintf(fpryo,"\t\t\toffset=%i \tbasereg=%x\n",j,basereg);
  i = (*pointer >> 21) & 0x1f;
  fprintf(fpryo,"\t\t\tbasereg=%%r%i\n",i);
#else
  fprintf(fpryo,"\n");
#endif
}

ryo_word (pointer,basereg)
int *pointer,basereg;
{
  int i,j,k;

  i = (*pointer >> 26) & 0x3f;

  if (i==0x1A) fprintf(fpryo,"1 "); /* store */
    else if (i==0x12) fprintf(fpryo,"0 "); /* load */
    else fprintf(fpryo,"error at %x\n",pointer);

  basic_instr(pointer,basereg);
}
```

References

[1] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications ACM*, 34(4):46–58, April 1991.

[2] M.D. Hill, J.R. Larus, A.R. Lebeck, M. Talluri, and D.A. Wood. Wisconsin Architectural Research Tool Set. *Computer Architecture News*, 21(4):8–10, September 1993.

[3] Ruby B. Lee. Precision Architecture. *Computer*, 22(1):78–91, January 1989.

[4] K. Patel, B.C. Smith, and L.A. Rowe. Performance of a Software MPEG Video Decoder. In *Proceedings ACM Multimedia 93*, pages 75–82, August 1993.

[5] Michael D. Smith. Tracing with pixie. ftp document, Center for Integrated Systems, Stanford University, April 1991.

[6] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Communications ACM*, 34(4):30–44, April 1991.

[7] Daniel Zucker and Ruby Lee. Reuse of High Precision Arithmetic Hardware to Perform Multiple Concurrent Low Precision Calculations. Technical Report No. CSL-TR-94-616, Computer Systems Laboratory, Stanford University, April 1994.