pipestat:
An Analysis Tool for Instruction Traces and sim_ppc Scrollpipe Files

Version 1.0 March 2018

Aaron Sawdey – IBM

1 Introduction

`pipestat` works with instruction traces. It distills and cross-references the information contained within them into an assembly-source centered report. It can read either a M1 scrollpipe file or a qtrace-format instruction trace. M1 scrollpipe files contain all the information that is in an instruction trace and adds cycle-by-cycle annotations of pipeline events that occurred during the execution of each instruction. Here are some of the general classes of analysis that pipestat performs:

- Annotated disassembly of the code executed in the trace or scrollpipe.
- Construct a control flow graph, identify loop constructs, and report hot loops and their location within the trace.
- Analyze branch behavior and compute Linear Branch Entropy metric for branches to identify hard-to-predict ones.
- "Top 10" lists showing the hot items for a number of metrics.

Within `pipestat`, instructions from the trace are tracked for long enough so that it can see instruction-to-instruction interactions, and then the information is aggregated into a per-instruction-address record. This allows `pipestat` to have reasonable runtime and memory size characteristics so you can use it on long traces (many billions of instructions). The following things are tracked to provide the annotations:

- What instruction addresses appear in the trace and how many times are they executed.
- What data addresses are used.
- What instruction execution (from trace/scrollpipe) last stored to a data address.
- What instruction last wrote to a register.
- What targets does a particular branch instruction have.
- Where do we come from when we come to this instruction.
- When was the first time each instruction address is seen in the trace.

1.0.1 Typographical conventions

In cases where there are long lines in examples or output, I have used a \ at the end of the line to indicate where I have broken them up to make the manual readable.

2 Using pipestat

2.1 Producing trace and pipe files

To use `pipestat` you will need to be able to create trace files and scrollpipe files.

2.1.1 Using `valgrind itrace` to produce a trace

Here is an example of how to trace something (an interesting little perl example) and then process that trace. The `vgi2qt` command converts the trace file into `qtrace` format which can be used with `pipestat` and `sim_ppc`.

```
valgrind --tool=itrace --binary-outfile=perlprime.vgi --dump-loadmap \
   --trace-extent=calltree --fnname=main --num-insns-to-collect=200m \
   perl -le 'map   print "Prime $_" if (1 x $_) !~ /^1?$|^(11+?)\1+$/  1..10001'
vgi2qt -f perlprime.vgi -o perlprime.qt
./pipestat -v -m perlprime.map perlprime.qt > out
```

In this example we are collecting a trace of everything after `main` is entered. The length of the trace is limited to 200 million instructions. This is long enough to get good statistics but not so long that running it through `pipestat` takes an excessive amount of time.

Here is a quick reference of useful `valgrind itrace` options:

`--tool=itrace`
: Tell valgrind to use the `itrace` tool.

`--help`
: Print all the generic `valgrind` options and also the options specific to `itrace`.

`--binary-outfile=file.vgi`
: Send itrace output to `file.vgi`.

`--dump-loadmap`
: Generate a load address map when the program is run in `valgrind`. The output file name is whatever you specified for `--binary-outfile` with the file extension (usually `.vgi`) replaced with `.map`. This file can then be given to `pipestat` so it can look up symbol and debug information to use when generating the annotated assembly output.

`--trace-extent=calltree|all|function`
: This option allows you to tell `itrace` how to limit what you want it to trace in the program. The three options are:

  `all` Trace the entire program starting from the very beginning.

  `calltree` Trace just a particular function and all other functions called from it.

  `function` Only trace a particular function, and do not trace any functions it calls.

  If you use `calltree` or `function`, the trace will have gaps where it skips over what happens betwen calls to the function being traced and child functions for `--trace-extent=function`. This doesn't affect `pipestat` too much but `sim_ppc` can produce misleading results on such traces if there are too many gaps.

`--fnname=func`
: This option specifies the function name to be traced with `--trace-extent=function` or `calltree`. If the function is a c++ function it can sometimes be necessary to use the `--demangle=no` option and specify a mangled c++ function name to `--fnname`.

`--num-insns-to-collect=NNm`
: Tell `itrace` to collect `NN` million instructions. This option understands suffixes `k` for thousand and `g` for billion.

The easiest way to get a version of `valgrind` that has the `itrace` functionality for tracing is to install the IBM Advance Toolchain, which can be found here:

https://developer.ibm.com/linuxonpower/advance-toolchain/

Once installed, you can add `/opt/atX.X/bin` to get the AT (Advance Toolchain) versions of `gcc`, `valgrind`, and `vgi2qt`. At the time of this writing, the current version is 11.0 so the path would be `/opt/at11.0/bin`.

2.1.2 Using `sim_ppc` to produce pipe files

The *IBM Power8 Performance Simulator for Linux on Power* `sim_ppc` can be found here:

https://developer.ibm.com/linuxonpower/sdk-packages/

Once installed, you can use the `run_timer` script to run a trace through the simulator:

```
/opt/ibm/sim_ppc/bin/run_timer_Power8 perlprime.qt 100000 10000 10000000 perlprime \
   -scroll_pipe 1 -scroll_begin 10000000 -scroll_end 10100000
```

This will produce the pipe file `perlprime.pipe` which can then be run through pipestat. In this example, we simulate 10,000,000 instructions to make sure that we are past initialization code and to get the internal state of the processor (caches, branch prediction, etc.) thoroughly warmed up. Then we run for a further 100,000 instructions producing the pipe file.

The command line arguments for `run_timer` are:

```
run_timer input.qt AAA BBB CCC output_name_tag [options]
```

Those initial arguments are required and must be given in that order.

`input.qt` This is the qtrace input. `sim_ppc` also understands how to read a compressed qtrace file if the filename ends with `.qt.gz` or `.qt.bz2`.

`AAA` The number of instructions to run and output counts and statistics for.

`BBB` The number of instructions between progress output that displays the current cycles per instruction.

`CCC` If greater than one, run this many instructions first to warm up the core model before running `AAA` instructions.

`output_name_tag`
This string is used to name the various output files from the simulator run.

The options for the `run_timer` script that come after the fixed ones are:

`-scroll_pipe 1`
Turn on scrollpipe output.

`-scroll_begin NNNN`
Start producing scrollpipe records after `NNNN` instructions. This count starts from the very beginning of the run, i.e. it includes the `CCC` instructions of warmup.

`-scroll_end MMMM`
Stop scrollpipe output after `MMMM` instructions, counting from the beginning of the trace.

The pipe file can be processed with `pipestat`:

```
pipestat -m perlprime.map -v perlprime.qt > perlprime.pstat
```

2.2 `pipestat` command line options

For reference, here is a description of all the command line options accepted by `pipestat`. You can also use `pipestat --help` to see a brief description of all the command line options.

`-o` *`file`* Send pipestat output to *file*. By default the output goes to stdout.

`-n` *N* Process only *N* instructions from the input file. If input is a scrollpipe file, `pipestat` processes input until the final event from the *N*th instruction is seen.

`-s` *S* Skip *S* records before starting. This option only applies to qtrace input.

`-h` *H* Set the history window size to *H* instructions. The history window determines how far back pipestat will keep full information on previous instructions so that it can find fusion pairs, Load-hit-store conflicts, and other instruction-to-instruction interactions. The default is 50,000 instructions.

`-m` *mapfile* Use load map file *mapfile* produced by valgrind itrace. This file tells pipestat where to locate the program and any dependent DSOs that were used to produce the trace, as well as the memory locations where they were loaded.

`-qt` The input file is a qtrace file. This is only needed if input is stdin or the input file name does not end in `.qt{,.gz,.bz2}`

`-v` Produce verbose output. A per-instruction and data address detail report is included in the output. For scrollpipe input you will also get a table for every instruction address showing the number of times it got each core event for each execution of that address.

`-traceRec` The trace record number of the first execution is printed at the beginning of each line of the annotated assembly listing. For qtrace input, the record number counts from the beginning of the file (i.e. it includes any instructions skipped with the `-s` option). For scrollpipe input, the record number is the architected instruction number from the M1.

`-inst` The 32-bit instruction is printed after the instruction address in each line of the annotated assembly listing. The instruction is always printed as a part of the `-v` output.

`-nhot` *N* *N* items will be reported and flagged in the annotated assembly listing for each of the hot reports. The default *N* setting is 20.

`-badLHS` *N* LHS with less than *N* instructions between store and load are considered bad. This determines which LHS interactions are considered for the hot list and also determines how far pipestat looks for stack parameter LHS. The default is 100 instructions.

`-LBEdepth` *N*
Use branch history depth *N* for Linear Branch Entropy calculations. This sets the number of previous branch decisions used to index the LBE tables. For the local versions the last *N* executions of the branch being predicted are used, and for the global version the last *N* branch decisions are used. The default is 20 branches.

`-badLarxLarx` *N*
Less than *N* instructions between two larx instructions is considered bad. This is used to flag larx-larx interactions. The default is 100 instructions.

`-debuginfo` *dir*
Add *dir* to the search path for finding debuginfo symbol files. You can specify multiple `-debuginfo` options to add multiple directories to the search path.

`-cfgDOT` *file*
Output the control flow graph to *file* in DOT format. See http://graphviz.org for more info and tools for visualizing DOT output.

`-ccp8` Simulate count cache for p8.

`-allLoops` Report all loops in the hot loop constructs report. Normally only *NHOT* will be reported.

`-d` Turn on huge quantities of debug output.

`-score` Output only a single line giving a code score metric. See [Code Score], page 21.

`-scorehdr` Output code score with a column header line as well.

`filename` Read input from *filename* instead of stdin. If the input file ends in .qt (optionally followed by .gz or .bz2) then the input is assumed to be qtrace format. Otherwise it is assumed to be scrollpipe. The default input is read from stdin, and is always assumed to be scrollpipe unless you have specified the -qt option.

It is recommended that you use the `-v` option to get the per instruction address detailed listing so you can see branch target listings. The `-n` option can be used to process only part of the trace. `Pipestat` also catches `SIGINT` so you you type `^C` it will stop reading the trace, report on what is has processed, and exit normally.

3 Pipestat output

When executed, `pipestat` output starts with a few informative sections:

- If you are using a map file and the DSO files used by the trace are available, the DSOs and their addresses and symbol count are listed. You can copy the DSO files referenced in the map file and edit the map file to reflect their new locations. It should work to copy trace, map file, and related DSOs to a different machine altogether but it should have the same architecture (for instance powerpc64 little endian) as the machine the traces were produced on.
- The number of instructions processed: `97723013 architected instructions processed`
- The number of unique instruction and data addresses encountered: `unique IA: 25568 unique DA: 512`
- Some statistics about the basic blocks that pipestat identified during execution are listed after the `Basic Block report`. A *Basic Block* is defined here as a sequence of instructions where control flow does not branch in or out in the middle of the block.

```
Loaded DSO /lib/powerpc64le-linux-gnu/ld-2.23.so at 0x4000f80 (31 symbols)
Loaded DSO /lib/powerpc64le-linux-gnu/libc-2.23.so at 0x4241d80 (2275 symbols)
Loaded DSO /lib/powerpc64le-linux-gnu/libm-2.23.so at 0x40f6400 (430 symbols)
Loaded DSO /lib/powerpc64le-linux-gnu/libpthread-2.23.so at 0x41e5100 (825 symbols)
Loaded DSO /lib/powerpc64le-linux-gnu/libcrypt-2.23.so at 0x43f0b00 (33 symbols)
Loaded DSO /lib/powerpc64le-linux-gnu/libdl-2.23.so at 0x40c0d80 (40 symbols)
Loaded DSO /opt/at11.0/lib/valgrind/vgpreload_core-ppc64le-linux.so \
   at 0x4060600 (60 symbols)
Loaded DSO /usr/bin/perl at 0x1001a600 (2083 symbols)
209715200 architected instructions processed
209715201 internal instructions processed
created 209715201 instStat objects 1614985607 searches 209715201 tag searches
end time 209715201 with 209715201 clock increments
annotations 1195555203 milestones 0
opened /proc/63826/status for VM info on pipestat run
VmHWM:    196480 kB
VmRSS:    196480 kB
VmData:   189248 kB
instStat: max concurrent 51021 current 50065 size 8+256 total 13154 kB
xrefData: max 41822 size 8+336 total 14050 kB
dxrefData: max 52701 size 8+80 total 4529 kB
inst_info: max 0 size 8 total 0 kB
da_offset_info: max 1180620 size 16+16 total 36895 kB
da_offset_info: max per xrefData 97398 avg 103.3 zero 30398 nonzero 11424
unique IA: 41822 unique DA: 52701
Basic Block report: 7858 blocks size stats: min 1i max 122i avg 5.32i \
   exec weighted avg. 5.31i
```

4 The “Top 10” reports

`pipestat` output begins with some “Top 10” style lists to try to identify the most interesting parts of the code. Use the `-nhot` option to change how many things it reports in each of these lists. The default is to report 20 items. Each list has a heuristic for ordering the tiems so that the most important are at the top of the list. The items displayed in some lists are also flagged in the annotated disassembly listing with a `HOT-XYZ` marker where `XYZ` is some short descriptor for the list. Here are the categories covered:

4.0.1 Hot execution count blocks

The `N:nnnn` notation means that this block is executed `nnnn` times; this is also used in the annotated disassembly section. Each instruction from each of these blocks will also be marked with `HOT-blk` after the `N:nnnn` in the annotated assembly. The list is ordered by number of executions times number of instructions in the block.

```
HOT execution count blocks:
  0x00001015b24c-0x00001015b264 N:2129905 7 inst trace inst 5292679
  0x000010156024-0x0000101560b0 N:359801 36 inst trace inst 5298161
  0x000010154ea8-0x000010154f28 N:283171 33 inst trace inst 5298310
  0x00001015b22c-0x00001015b238 N:2134873 4 inst trace inst 5292671
```

4.0.2 Hot misaligned short loops

This lists single-block loops, ordered by the number of iterations times a weighting factor that depends on how the loop is misaligned (crossing 32B or 128B) and whether it is shorter than 128B overall. This section also includes a table of loop sizes for single-block loops which gives some data on how many loops of each size were found and how much they were executed.

```
HOT misaligned short loops:
  0x00001002e11c-0x00001002e14c N:10000 13 inst short misalign32
  0x0000042cde70-0x0000042cde98 N:950 11 inst short misalign128
  0x0000042c91e0-0x0000042c9200 N:59 9 inst short misalign128
  0x0000042d3160-0x0000042d3180 N:44 9 inst short misalign128
  0x0000042cdfb0-0x0000042cdfc4 N:37 6 inst short misalign32
  0x0000042b6638-0x0000042b6640 N:2 3 inst short misalign32
Loop size summary data:
Instructions  loops  total iter  min iter   max iter   avg iter  total inst  % of trace
           2      1          27        27         27      27.00          54        0.00
           3      7        2442         2       2336     348.86        7326        0.00
           4      3       20344        11      20207    6781.33       81376        0.04
           5      2          16         7          9       8.00          80        0.00
           6      1          37        37         37      37.00         222        0.00
           7      1         220       220        220     220.00        1540        0.00
           8      5        3325         3       2532     665.00       26600        0.01
           9      2         103        44         59      51.50         927        0.00
          11      4        1173         1        950     293.25       12903        0.01
          13      1       10000     10000      10000   10000.00      130000        0.06
```

The loop from 0x1002311c to 0x1002314c spans four 32 byte blocks (0x10023110-0x10023120, 0x10023120-0x10023130, 0x10023130-0x10023140, and 0x10023140-0x10023150). Adding a single no-op instruction before the loop would shift the loop's address range to 0x10023120-0x10023150, such that it would span only three 32 byte blocks. Thus it is classified as `misalign32`.

```
0x00001002e11c M>ld           r7,-8(r9)          N:10001 HOT-blk \
   Use:3 LoopHdr 13i 10000 iter 130000i dyn short misaligned 32B loop 13i \
   from:0x1002e14c DA ref from inter 0x100d66d8 16177i 0.50%
0x00001002e120   addi         r6,r9,-16          N:10001 HOT-blk Use:10
0x00001002e124   addi         r9,r9,-8           N:10001 HOT-blk Use:4
0x00001002e128   lwz          r8,12(r7)          N:10001 HOT-blk Use:1 \
```

```
   DA ref from inter 0x100f01c8 16103i 0.50%
0x00001002e12c   oris         r8,r8,0x2          N:10001 HOT-blk Use:1
0x00001002e130   stw          r8,12(r7)          N:10001 HOT-blk HOT-LHS \
   DA ref to intra 0x1002e138 2i 100.00%
0x00001002e134   ld           r7,0(r9)           N:10001 HOT-blk Use:1 \
   DA ref from inter 0x100d66d8 16183i 0.50%
0x00001002e138   lwz          r8,12(r7)          N:10001 HOT-blk Use:1 \
   HOT-LHS DA ref from intra 0x1002e130 2i 100.00% redundant: intra: 10001
0x00001002e13c   oris         r8,r8,0x801        N:10001 HOT-blk Use:1
0x00001002e140   stw          r8,12(r7)          N:10001 HOT-blk
0x00001002e144   ld           r10,16(r28)        N:10001 HOT-blk Use:1
0x00001002e148   cmpld        cr7,r10,r6         N:10001 HOT-blk Use:1
0x00001002e14c C ble          cr7,0x1002e11c     N:10001 HOT-blk \
   Loop backedge to:0x1002e11c br prob:99.99% \
   (avg seq tk 10000.0 ntk 1.0 LBE 0.000/0.000)
```

4.0.3 Hot Loop constructs

In order to identify loops with control flow such as `if` statements inside them, `pipestat` builds a Control Flow Graph (CFG) of the code seen in the trace or scrollpipe input. Each node in this graph is a basic block, a sequence of instructions that is executed linearly. The arcs or edges in the graph represent the places where the next instruction after the end of the basic block can be found. If the block ends in a conditional branch, then there will be two edges, one for the case where the branch was not taken and execution proceeds to the next block of code, and another for the case where the branch is taken and execution proceeds to the branch target.

Once we have the CFG built for the code in the trace, we have to have some way of finding loops. The way to do this is to observe that even if there are `if` statements or other conditional statements in a loop, the only way to get to any block in the loop is to go through the first block in the loop. This property where you can only reach the blocks inside the loop by first executing the block at the beginning of the loop is called *dominance*. More carefully stated, node **A** dominates node **B** if every path from the beginning of the program to **B** must first pass through **A**.

We find a loop by finding the CFG edge that goes back from the end of the loop to the top of the loop. Specifically we look for a branch where the node we are going to dominates the node we just came from. The destination node is the loop header block. Then we identify all other nodes in between the loop header and the source of the backedge, which must also be dominated by the loop header.

Currently I don't make a distinction between bl/blr and other branches so these *loop constructs* may in fact span across functional boundaries, hence the bl/blr count in the last column. Sometimes recursive functions can be identified as loops so a series of recursive calls would have more bl than blr, and a series of recursive returns would have more blr than bl.

The taken/untaken branch statistics are for branches other than the loop backedges, and they are normalized by the iteration count. If the command line option `-allLoops` is specified, all detected loop constructs are printed, otherwise the number is controlled by `-nhot`.

The columns in the loop constructs table are:

`Header blk IA`
: is the instruction address of the beginning of the header block. This is the block that dominates all other blocks in the loop construct, and loop back-edge branches go to this block. The back-edge can also be a fallthrough if the compiler put the last block in the loop right before the header block.

`static`
: The count of the number of instructions in the blocks that are part of this loop.

`dynamic`
: The total number of instructions executed in this loop in the trace.

`iter` The number of times the loop header is executed. Because the other blocks in the loop can only be reached through the header, this is a good estimate of the number of loop iterations.

`inst/iter` The dynamic instruction count divided by `iter`.

`nodes` The number of blocks in the loop.

`taken` The number of taken branches per iteration.

`untaken` The number of un-taken branches per iteration.

`BkEdge:` This just labels that the columns following are branch stats for back edge branches only.

`Edge:` The columns following apply to non-backedge branches or transitions.

`bd_tk` The number of branch-decrement type branches taken. These are all branches that branch based on CTR even if they are also looking at a CR field.

`bd_nt` The number of branch-decrement type branches not taken.

`bc_tk` The number of conditional (CR only) branches taken.

`bc_nt` The number of conditional branches not taken.

`br` The number of unconditional branches.

`fallth` The number of fallthrough block transitions. Since we track un-taken branches separately, this is really the number of times we went through a block that ends without a branch because the following instruction was a branch target from somewhere else.

`BL` The number of branch-and-link instructions.

`BLR` The number of branch-to-LR instructions. Sometimes when a recursive function returns multiple times consecutively, it is identified as a loop. But if the number of BL/BLR are equal and not zero for a loop construct, it's likely that the loop calls a function that is not used anywhere else. If the called function is used elsewhere, those blocks won't be dominated by the loop header and so won't end up as part of the loop construct.

HOT Loop constructs (123 total)

Header blk IA	arch inst	static	dynamic	iter	inst/iter	nodes	taken	untaken	BkEdge: bd_tk	bd_nt	bc_tk	bc_nt	br	fallth	Edge: bd_tk	bd_nt	bc_tk	bc_nt	br	fallth	BL	BLR
0x000000001015b22c	5292671	485	75435158	2134873	35.3	101	2.93	3.57	0	0	0	0	684721	0	0	40850	3992932	7580848	2262177	51488	324145	79210
0x000000000101212c8	581627	36	1889662	118251	16.0	7	1.01	1.37	0	0	27	0	0	0	0	0	118251	162113	1397	41766	0	0
0x00000000042cb650	181343	15	1131372	116523	9.7	6	0.00	3.40	40764	0	0	0	0	0	0	140055	0	256578	0	0	0	0
0x00000000042b430c	131602	6	309132	51522	6.0	2	1.00	0.00	0	0	0	51522	0	0	0	0	51522	0	0	0	0	0
0x000000001002e11c	3373180	13	130000	10000	13.0	1	0.00	0.00	0	0	10000	0	0	0	0	0	0	0	0	0	0	0

4.0.4 Hot long latency blocks

Blocks with long latency instructions in them. These are marked in the annotated disassembly listing with `HOT-LongLat`. `pipestat` has a builtin list of long latency instructions. Generally they are instructions that take more than 5 cycles to complete or have other restrictions that may cause them to take a long time.

```
HOT long latency instruction blocks:
  0x00001015b24c-0x00001015b264 N:2129905 7 inst badness 1521361
  0x00001001b110-0x00001001b11c N:320010 4 inst badness 400012
  0x0000042cb620-0x0000042cb62c N:231277 4 inst badness 289096
  0x000010156024-0x0000101560b0 N:359801 36 inst badness 139923
  0x0000042cb884-0x0000042cb888 N:53963 2 inst badness 134908
```

In this example, 0x1015b24c is being flagged because the block is hot and contains an `mtctr` instruction.

```
0x00001015b24c f addis        r9,r2,-11          N:2129905 HOT-blk Use:1 TOC
0x00001015b250 f addi         r9,r9,13160        N:2129905 HOT-blk Use:2 TOC
0x00001015b254   rldicr       r8,r24,2,61        N:2129905 HOT-blk Use:1
0x00001015b258   lwax         r8,r9,r8           N:2129905 HOT-blk Use:1
0x00001015b25c   add          r9,r8,r9           N:2129905 HOT-blk Use:1
0x00001015b260   mtctr        r9                 N:2129905 HOT-blk Use:1 \
   HOT-LongLat LBAD 5
0x00001015b264 U bctr                           N:2129905 HOT-blk \
   to: 20 targets
```

This is a case where there are too many branch targets to be listed inline. However if you specified the `-v` option, you can find this output later in the file:

```
IA 0x00001015b264 image 0x4e800420 bctr \
   execution_count 2129905
Branches to: 0x1015b480:78402 0x1015b850:243007 0x1015bbe8:78963 0x1015bd84:243691 \
   0x1015be84:684 0x1015c318:78963 0x1015c890:243006 0x1015c950:243007 \
   0x1015c9ac:39481 0x1015ca0c:39481 0x1015d71c:684 0x1015d9a4:359118 \
   0x1015daa0:40041 0x1015dafc:39481 0x1015ddc0:684 0x1015e02c:40041 \
   0x1015e1cc:685 0x1015e1ec:684 0x1015e230:359118 0x1015ec28:684
```

4.0.5 Hot redundant loads

A redundant load is the the situation where register **X** is stored into memory location **A** and then later **A** is loaded back into **X** with no intervening changes to **A** or **X**. For the code path taken in the trace, the store and load of **X** could have been skipped altogether. Often this indicates that more registers were saved/restored than needed to be because there was an early exit test and the function could have benefited from *shrink-wrapping*. This optimization defers saving register values so that an early exit from the function does not unnecessarily save and restore registers that never ended up being used. This list gives some overall statistics on redundant loads based on what type of memory location was involved (stack or non-stack) and whether the store and load were separated by a `bl` instruction (intra-procedural if they were not, inter-procedural if they were).

```
HOT redundant loads: intra+stack: 945618 (0.45%) inter+stack: 14040 (0.01%) \
   intra: 135341 (0.06%) inter: 12854 (0.01%)
  0x000010156024-0x0000101560b0 N:359801 redundant loads 359501
  0x000010121440-0x0000101214a0 N:41118 redundant loads 242791
  0x00001015952c-0x000010159568 N:40974 redundant loads 81948
  0x0000042b1b40-0x0000042b1b9c N:5643 redundant loads 76910
  0x0000042cbbf0-0x0000042cbc28 N:65832 redundant loads 48538
  0x000010121794-0x0000101217a0 N:39614 redundant loads 39614
```

4.0.6 Hot bad branch hints

This section reports on branches that have the hint bits set but did not go the indicated direction at least 90% of the time. You can find these in the annotated disassembly by searching for `HOT BAD HINT`. Recent versions of `gcc` do not generally set the hint bits unless you are using profile-feedback optimization, so this is usually limited to assembly code that explicitly sets them, or code from older compilers.

```
HOT bad branch hints:
0x0000042cb494 hint likely not taken but was taken 55.20% (2132/3862)
0x0000101683ec hint likely taken but was not taken 100.00% (685/685)
0x0000101673c8 hint likely not taken but was taken 50.00% (685/1370)
0x0000101674b0 hint likely not taken but was taken 100.00% (124/124)
```

4.0.7 Hot branch mispredict count

This section reports on branches that are mispredicted the most number of times. This is only available if the input was a scrollpipe file as this relies on the mispredicted branch event from `sim_ppc`. These branches are marked with `HOT-Mispred`.

```
branch mispredict summary 16267 branches 230 mispredict 227 penalty samples avg penalty 19.1cy total penalty 4328cy
HOT branch mispredict count:
0x00001015b510 mispredicted   138 times ( 98.6% of   140) pen  17.0cy avg over   138 LBE 0.9857/0.0167
0x00001015b264 mispredicted     8 times (  0.9% of   861) pen  24.8cy avg over     8 LBE 0.0000/0.0000
0x0000100d7fb4 mispredicted     7 times ( 28.0% of    25) pen  31.0cy avg over     7 LBE 0.0000/0.0000
0x0000042cb63c mispredicted     4 times (  3.8% of   106) pen  17.0cy avg over     4 LBE 0.0755/0.0233
0x00001015da54 mispredicted     3 times (  2.8% of   108) pen  17.7cy avg over     3 LBE 0.0185/0.0455
0x00001015b238 mispredicted     3 times (  0.3% of   873) pen  17.0cy avg over     3 LBE 0.0046/0.0047
```

4.0.8 Hot branch mispredict frequency

This section reports on branches that are mispredicted the most frequently, and like the mispredict count table, it is only available if the input was a scrollpipe. These branches are also marked with `HOT-Mispred`.

```
HOT branch mispredict frequency:
0x0000042afe74 mispredicted 100.0% of 1 pen 21.0cy avg over 1
0x00001015b510 mispredicted 98.6% of 140 pen 17.0cy avg over 138
0x0000100f1104 mispredicted 66.7% of 3 pen 19.0cy avg over 2
0x0000042b3d5c mispredicted 60.0% of 5 pen 21.0cy avg over 3
0x0000042b1d40 mispredicted 50.0% of 4 pen 89.0cy avg over 2
```

4.0.9 Hot branches with high linear branch entropy and executed frequently

This section lists branches that have a high Linear Branch Entropy (see S. De Pestel et. al., "Linear Branch Entropy: Characterizing and Optimizing Branch Behavior in a Micro-Architecture Independent Way," *IEEE Transactions on Computers* vol. 66, no. 3, pp. 458-472). The two numbers listed after `LBE` are the global and local entropy. The global entropy calculation is based on the sequence of taken/not-taken branches that came before and may have been at different instruction addresses. The local entropy calculation is based only on the previous history of decisions at the branch address given. These branches are marked with `HOT-LBE`.

```
HOT branches with high linear branch entropy and executed frequently
  0x0000042cb63c N:231263   LBE 0.9641/0.0058
  0x00001015b510 N:78650    LBE 0.9897/0.0058
  0x0000042cb88c N:39452    LBE 0.9889/0.0516
  0x0000042cb8a0 N:19942    LBE 0.9915/0.0203
  0x0000042cba34 N:14511    LBE 0.9667/0.0258
```

4.0.10 Hot load hit store separated by less than 100 instructions

This section reports on code paths where a store to a data address was loaded less than 100 instructions later. This threshold can be changed with the command line option `-badLHS`. Both the load and store are marked with `HOT-LHS` in the annotated disassembly. Each line reports on a single pair of instruction addresses.

```
HOT load hit store separated by less than 100 instructions:
                                  of all exec:     Pathlength         Std. redund other   AGEN
Store IA       Load IA         Count   %     count min count   Avg max    Dev.  loads store registers     Store Values
0x000010156038 0x000010156044 359801 100.0% 359801  3 359801  3.00   3    0.0       0 1/1/1 st: G31 ld: G31
0x00001015606c 0x00001015607c 359801 100.0% 359801  4 359801  4.00   4    0.0       0 1/1/1 st: G31 ld: G31
0x000010156050 0x000010156060 359801 100.0% 359801  4 359801  4.00   4    0.0       0 1/1/1 st: G31 ld: G31
0x000010154ed0 0x000010154ee4 283171 100.0% 283171  5 283171  5.00   5    0.0       0 1/1/1 st: G3 ld: G3
0x00001015b0ec 0x00001015b0f4  39606 100.0%  39606  2  39606  2.00   2    0.0       0 0/0/0 st: G1 ld: G1
0x000010154ef0 0x000010154f08 283171 100.0% 283171  6 283171  6.00   6    0.0       0 1/1/1 st: G3 ld: G3
Total LHS events: 10349663 21.0% of loads 42.8% of stores
```

The columns and additional flags at the end are as follows:

`Store IA` The instruction address of the store.

`Load IA` The instruction address of the load.

`Count` The number of times this store and load IA referred to the same memory address with no intervening changes to that memory address.

`% of all exec:`
The percentage of all executions of this load IA that got data from the store IA.

`count` The total execution count of the load IA.

`Pathlength min`
The minimum instruction pathlength between the store and load.

`count` The number of times the minimum path occurred.

`Avg` The average instruction pathlength between the store and load.

`max` The maximum instruction pathlength between the store and load.

`Std. Dev.` The standard deviation of the pathlength between store and load.

`redund loads`
The number of times the load was redundant, i.e. the load is into the same register as the store wrote, and no intervening instruction changed the value of that register, meaning the load could have been skipped.

`other store`
The min/avg/max number of other unrelated stores in between the store and load.

`AGEN registers`
The registers used to generate the store and load address.

`disjoint` The store does not write all the bytes being read by the load.

`BASE REG CHANGED: nn`
The store and load had the same base register but the value of that base register changed in between so we got a LHS `nn` times even though the displacements were different.

`Store values`
If the input was a qtrace file with register data values you may see data such as this:

```
v=0: 12494 93% v=1: 451 3% v=-8..7: 13014 97% v=0..255: 13089 98%
 v=-32768..32767: 13381 100%
```

This breaks down as follows:

`v=0: 12494 93%`
The store value was zero 12494 times or 93% of all the LHS occurrences.

`v=-8..7: 13014 97%`
The store value was in the range -8 to 7 a total of 13014 times or 97% of all the LHS ocurrences.

4.0.11 Hot load hit store related events from scrollpipe

If `pipestat` is processing a scrollpipe file, there are a number of events from `sim_ppc` that indicate that load/store interference of some kind has taken place. Both the load and store are marked with `HOT-LHS` in the annotated disassembly listing.

```
HOT load hit store related events from scrollpipe
sf_sg event_store_flush_hit_load_same_group
sf_dg event_store_flush_hit_load_diff_group
lf_sg event_load_flush_load_hit_store_same_grp
fw_sg event_load_hit_store_forward_same_grp
fw_dg event_load_hit_store_forward_diff_grp
sffin event_store_forward_finish
SHLtb event_load_hit_SHL_table_no_issue
rj_sg event_load_reject_load_hit_store_same_grp_no_sdq
rj_nr event_load_reject_LHS_store_not_ready
rj_dg event_load_reject_load_hit_store_diff_grp_no_sdq
rj_nc event_load_reject_load_hit_store_diff_grp_not_cont
rj_EA event_load_reject_load_hit_store_partial_EA_cmp
Address          Mnemonic    Exec  Path min/avg Total sf_sg sf_dg lf_sg fw_sg fw_dg sffin SHLtb rj_sg rj_nr rj_dg rj_nc rj_EA
0x00001015e5e8 lwa             106    15    15.0   884     0     0     0     0   106   106     0     0     3   669     0     0
0x00001015607c lwa             109     4     4.0   774     0     0     0     0   109   109     0     0     0   556     0     0
0x000010156060 lwa             109     4     4.0   551     0     0     0     0   111   109     0     0     0   331     0     0
0x00001015bcd4 ld              191    47    55.6   382     0     0     0     0   191   191     0     0     0     0     0     0
0x00001015bcdc lwz             189    51    60.4   379     0     0     0     0   189   189     0     0     0     1     0     0
0x000010156044 lwa             104     3     3.0   332     0     0     0     0   104   104     0     0     0   124     0     0
```

The output itself lists the names of the pipe events. Here are the definitions of the remaining fields:

`Address` The address of the load instruction. The annotated assembly output for the load will have the address of the most frequent related store. The output provided by `-v` will have a listing of all store addresses that provide data to this load.

`Mnemonic` The instruction mnemonic for the load instruction.

`Exec` The number of executions of this instruction address that had a load-hit-store related event.

`Path min` The minimum number of instructions between the store and this load.

`avg` The average number of instructions between the store and this load.

`Total` The total number of load-hit-store related events recorded for all executions of this instruction address.

4.0.12 Hot register def to use with long latency and few instructions

When `pipestat` is processing scrollpipe input, it can determine the number of cycles between the issue cycle of one instruction and another that follows later. In this case `pipestat` is using its register tracking to find a pair of instructions where the second one uses a result from the first. Because we know both the number of instructions separating them and the number of cycles (counting issue-to-issue), we can compute cycles per instruction (CPI) for that sequence. This list is sorted by that CPI multiplied by the number of times the sequence was executed. The instructions on this list are marked with `HOT-def-use` in the annotated disassembly listing.

```
HOT register def->use with long latency and few instructions
0x00001015608c N:109   65.3 cyc  1.0 inst  65.31 cyc/inst
0x00001015b254 N:861    7.6 cyc  1.0 inst   7.56 cyc/inst
0x00001015b22c N:873    6.1 cyc  1.0 inst   6.06 cyc/inst
0x00001015b250 N:861   11.0 cyc  2.0 inst   5.51 cyc/inst
0x000010156074 N:109   42.6 cyc  1.0 inst  42.56 cyc/inst
0x00001015b240 N:861    4.7 cyc  1.0 inst   4.68 cyc/inst
```

4.0.13 Verbose mode output

The `-v` option adds an additional section of output near the end. This is a dump of the information that `pipestat` has gathered about each instruction address (IA) encountered in the input.

4.0.13.1 Instruction information and data linkage

```
IA 0x0000042afec4 image 0xe92e0068 ld          r9,104(r14) \
   execution_count 4 DA ref min 32i mode 32i average 6921.8i from \
   other IA: intra/32i/0x42afec8:1 intra/106i/0x42afec8:1 \
   inter/13293i/0x42b1ebc:1 inter/14256i/0x42b1ebc:1
```

Here is a breakdown of the information present in this first line.

`IA 0x0000042afec4`
: The memory address of this instruction.

`image 0xe92e0068`
: The 32-bit instruction present at this memory address.

`ld r9,104(r14)`
: The disassembly of the instruction.

`execution_count 4`
: The number of times this instruction was executed.

`DA ref min 32i mode 32i average 6921.8i`
: This instruction was connected to one or more other instructions by a data access. The minimum path of execution between them was 32 instructions and the mode of all the possible paths was also 32 instructions. The average number of instructions was 6921.8.

`from other IA:`
: Here we have `from` because the other instruction was a store that preceded this load. If it had been the other way around and this was a store, the word `to` would appear here instead.

`intra/32i/0x42afec8:1`
: These blocks describe individual other instructions that form the data access link. In this case `intra` indicates that there was not a `bl` or `blr` instruction between them, meaning they are in the same function call. The `32i` indicates this path was 32 instructions in length. `0x42afec8` is the address of the other instruction, and `:1` is the count of the number of times this path occurred, i.e. only once.

`inter/13293i/0x42b1ebc:1`
: This is an example of a link with a much longer path, and `inter` means there is at least one `bl`/`blr` on the path, so the other instruction is in a different function call context.

`/LARXH` This instruction and the previous instruction are both `larx` type instructions to the same address, which can cause performance issues if they are too close.

`/LARXF` This instruction and the previous instruction are both `larx` type instructions and the distance between them is less than 100 instructions or the amount specified by the `-badLarxLarx` option.

4.0.14 Branch sources and targets

```
Branches from: 0x5d6add8:342 0x5d6adf4:36
Branches to: 0x5156870:1 0x1026229c:1
```

This section lists any branch sources that branch to this address, followed by any branch targets that this instruction branches to (if it is a branch). In both cases the list is a space-separate series of `ADDR:count` pairs.

4.0.14.1 Event count table

This table has a row for each time the instruction address was executed. The `IOP` number is the internal operation number from the scrollpipe and can be used with other scrollpipe tools to find this point in the scrollpipe file. For each execution, the number of cycles from issue to finish is `cyc`. The remainder of the columns counts of each scrollpipe event. The header row has event numbers that can be looked up in the `.event_type` section near the beginning of the input file. Under those columns, each row of the table has the number of times that event ocurred for that execution of the instruction.

Events:

IOP	cyc	2	3	5	7	8	20	41	42	46	72	151	312	900	902	1002	1004
12059283	9	1	1	6	1	1	1	1	1	0	2	0	0	0	1	0	1
12059323	16	1	1	6	1	1	1	1	1	1	4	1	0	0	1	0	1
12085514	9	2	2	12	2	2	1	2	2	0	4	0	1	1	1	1	1
12085639	9	2	1	6	1	1	1	1	1	0	2	0	0	0	1	0	1

5 Annotated disassembly

```
Inst annotated disassembly
/usr/bin/perl:Perl_start_glob
0x000010154ea8 > lwa          r9,48(r3)          N:283171 HOT-blk Use:2 from:0x1015c96c,0x1015c9c8 DA ref from inter 0x10154f14 179i 82.63% 124i min
0x000010154eac   ld           r7,40(r3)          N:283171 HOT-blk Use:8 DA ref from inter 0x1011f9b0 164i 5.38%
0x000010154eb0   addi         r10,r9,-1          N:283171 HOT-blk Use:2
0x000010154eb4   addi         r9,r9,-2           N:283171 HOT-blk Use:4
0x000010154eb8   extsw        r8,r10             N:283171 HOT-blk Use:2
0x000010154ebc   stw          r10,48(r3)         N:283171 HOT-blk
0x000010154ec0   rldicr       r8,r8,3,60         N:283171 HOT-blk Use:3
0x000010154ec4   extsw        r10,r9             N:283171 HOT-blk Use:1
0x000010154ec8   rldicr       r10,r10,3,60       N:283171 HOT-blk Use:5
0x000010154ecc   ldx          r8,r7,r8           N:283171 HOT-blk Use:2 DA ref from inter 0x10156090 137i 16.22% 131i min
0x000010154ed0   stw          r9,48(r3)          N:283171 HOT-blk HOT-LHS DA ref to intra 0x10154ee4 5i 100.00%
0x000010154ed4   srdi         r8,r8,6            N:283171 HOT-blk Use:1
0x000010154ed8   cmpdi        cr7,r8,3           N:283171 HOT-blk Use:20
0x000010154edc   lwzx         r9,r7,r10          N:283171 HOT-blk Use:1 DA ref from inter 0x10156078 147i 16.22% 141i min
0x000010154ee0 M stw          r9,116(r4)         N:283171 HOT-blk
0x000010154ee4   lwz          r9,48(r3)          N:283171 HOT-blk Use:1 HOT-LHS DA ref from intra 0x10154ed0 5i 100.00%
0x000010154ee8   addi         r9,r9,-1           N:283171 HOT-blk Use:1
0x000010154eec   extsw        r10,r9             N:283171 HOT-blk Use:2
0x000010154ef0   stw          r9,48(r3)          N:283171 HOT-blk HOT-LHS DA ref to intra 0x10154f08 6i 100.00%
0x000010154ef4   rldicr       r9,r10,3,60        N:283171 HOT-blk Use:2
0x000010154ef8   ld           r10,40(r3)         N:283171 HOT-blk Use:1 DA ref from inter 0x1011f9b0 183i 5.39%
0x000010154efc   lwzx         r9,r10,r9          N:283171 HOT-blk Use:1 DA ref from inter 0x1015605c 162i 16.22% 156i min
0x000010154f00 M stw          r9,112(r4)         N:283171 HOT-blk HOT-LHS DA ref to intra 0x10154fec 11i 100.00%
0x000010154f04   ori          r2,r2,0x0          N:283171 HOT-blk NOP
0x000010154f08   lwz          r9,48(r3)          N:283171 HOT-blk Use:1 HOT-LHS DA ref from intra 0x10154ef0 6i 100.00%
```

If you use the `-traceRec` command line option, the number at the beginning of the command line tells you what the trace record number was when that instruction address was first encountered. This can be used to determine where to start scrollpipe output in a `sim_ppc` run.

```
00000659 0x0000041eb448 M std          r31,-8(r1)         L:64 N:6 DA ref to intra 0x041eb4c4 31i 83.33%
00000660 0x0000041eb44c M std          r30,-16(r1)        L:64 N:6 DA ref to intra 0x041eb4c0 29i 83.33%
00000661 0x0000041eb450   cmpdi        cr7,r8,0           L:75 N:6 Use:4
00000662 0x0000041eb454   mr           r31,r3,r3          L:75 N:6 Use:9.7
00000663 0x0000041eb458   std          r0,16(r1)          L:64 N:6 DA ref to intra 0x041eb4bc 25i 83.33%
00000664 0x0000041eb45c   stdu         r1,-48(r1)         L:64 N:6 Use:24.8
00000665 0x0000041eb460 C bne          cr7,0x41eb4e0      L:75 N:6 to:0x41eb4e0 br prob:16.67% (avg seq tk 1.0 ntk 5.0 LBE 0.000/0.000)
00248164 0x0000041eb464   li           r10,0              L:80 N:5 Use:2
00248165 0x0000041eb468   li           r8,1               L:80 N:5 Use:5
00248166 0x0000041eb46c   clrldi       r10,r10,32         L:80 N:5 Use:2
00248167 0x0000041eb470   lwarx        r9,0,r31           L:80 N:5 Use:1 LBAD 3 DA ref from inter 0x041ed26c 2396i 33.33%
00248168 0x0000041eb474   subf.        r9,r10,r9          L:80 N:5 Use:1
00248169 0x0000041eb478 C bne          0x41eb484          L:80 N:5
00248170 0x0000041eb47c   stwcx.       r8,0,r31           L:80 N:5 Use:1 LBAD 3 DA ref to intra 0x041ed268 140i 80.00%
```

Here is another example of the output when `pipestat` input is a scrollpipe file.

```
0x00001015b480 > ld           r8,296(r1)         N:140 Use:1 3cy from:0x1015b264,0x1015b5e4 \
   DA ref from 1.4%-inter 0x101602bc 110i 48.92% 104i min 52.4cy cy:9,11,25
0x00001015b484   cmpdi        cr7,r8,0           N:140 Use:1 2cy cy:10,12,26
0x00001015b488 U beq          cr7,0x1015b490     N:140 to:0x1015b490 cy:11,13,27
0x00001015b490 > lwa          r4,32(r30)         N:140 Use:2 3cy from:0x1015b488 HOT-LHS \
   DA ref from 1.4%-inter 0x1015bc10 77i 97.14% 73i min 33.9cy cy:10,20,39
0x00001015b494   lwz          r9,48(r28)         N:140 HOT-def-use Use:1 12cy \
   DA ref from inter 0x101207e8 173i 48.55% 167i min 106.8cy cy:11,12,21
0x00001015b498   cmpw         cr7,r9,r4          N:140 Use:1 2cy cy:12,23,43
0x00001015b49c C bgt          cr7,0x10160328     N:140 cy:13,24,44
0x00001015b4a0   lwz          r7,112(r16)        N:140 Use:2 9cy \
   DA ref from 50.4%-inter 0x1015b4ec 128i 48.92% 122i min 86.1cy redundant: intra: 69 cy:9,10,24
0x00001015b4a4   lwz          r9,24(r30)         N:140 Use:1 9cy DA ref from 1.4%-inter 0x1015bbf8 88i 97.14% 84i min 32.6cy cy:9,13,23
0x00001015b4a8   cmplw        cr7,r9,r7          N:140 Use:1 3cy cy:9,16,31
0x00001015b4ac U bge          cr7,0x1015b4ec     N:140 to:0x1015b4ec cy:10,17,33
0x00001015b4ec > stw          r7,112(r16)        N:140 from:0x1015b4ac DA ref to intra 0x1015bbe8 37i 98.59% 35i min 69.3cy cy:9,12,30
0x00001015b4f0   lwz          r9,28(r30)         N:140 Use:1 9cy DA ref from 1.4%-inter 0x1015bc08 88i 97.14% 84i min 36.0cy cy:9,13,25
0x00001015b4f4   stw          r9,116(r16)        N:140 DA ref to intra 0x1015bc00 41i 98.59% 39i min 36.4cy cy:12,20,35
0x00001015b4f8   ld           r18,40(r30)        N:140 Use:1 8cy DA ref from 1.4%-inter 0x1015bc04 91i 97.14% 87i min 39.4cy cy:9,14,23
0x00001015b4fc   cmpdi        cr7,r18,0          N:140 Use:1 2cy cy:10,17,31
0x00001015b500 C beq          cr7,0x10160bf8     N:140 cy:10,17,32
0x00001015b504   lbz          r9,1(r18)          N:140 Use:1 4cy cy:11,18,29
0x00001015b508   andi.        r9,r9,0xdf         N:140 Use:1 2cy cy:12,20,32
0x00001015b50c   cmpwi        cr7,r9,30          N:140 Use:1 2cy cy:14,22,33
0x00001015b510 C beq          cr7,0x101602b8     N:140 \
   HOT-LBE HOT-Mispred: 98.6% 17.0cy to:0x101602b8 cy:15,23,34 br prob:50.71% (avg seq tk 1.0 ntk 1.0 LBE 0.986/0.017)
0x00001015b514   li           r9,0               N:69 Use:1 2cy cy:7,7,7
0x00001015b518   std          r9,296(r1)         N:69 cy:11,11,11
0x00001015b51c   ori          r2,r2,0x0          N:69 NOP cy:55,58,65
0x00001015b520 > cmpdi        cr7,r17,0          N:317 Use:1 3cy \
   from:0x1015bd88,0x1015bf00,0x1015c320,0x1015c8b4,0x1015c9ec,0x1015ca24,0x1015da54 cy:7,9,31
0x00001015b524 U beq          cr7,0x1015bca0     N:317 to:0x1015bca0 cy:8,10,32
0x00001015b530 U>beq          cr7,0x1015b880     N:1 from:0x1015ca34 to:0x1015b880 cy:7,7,7
0x00001015b534 > ld           r10,280(r28)       N:2 Use:2 3cy from:0x1015b87c DA ref from inter 0x10156194 7037i 50.00% 4336.5cy cy:9,9,9
0x00001015b538 U b            0x1015b564         N:2 to:0x1015b564 cy:5,6,6
```

Here is a key to the annotations:

- When a function boundary is crossed, you will see a line with just `file_name:function_name`.
- The first column is the instruction address from the trace
- In between the address and the disassembly, there may be one or more flag characters:

  `>` This address was a branch target. The annotation section on the right will have a `from:0xnnnnn` to tell you where the code jumped to this address from. Up to 8 addresses will be listed in the annotation. If there are more, it will simply say `from: nn targets`. The `-v` option will enable the verbose per-instruction-address output following the annotated assembly listing, which will list all branches that target the address.

  `U` This branch was effectively unconditional and branched somewhere every time it was executed. Destination addresses will be marked with a `to:0xnnnnn` annotation. Similar to the `>` annotation, up to 8 destinations will be listed inline, beyond that you need to use the `-v` option to get the full list of branch destinations for this branch.

  `C` This branch was conditional and did not branch every time. Up to 8 destination addresses will be listed in a `to:0xnnnnn` annotation, beyond that `-v` will be needed to see a full list in the separate per-instruction-address detailed output.

  `F` This instruction has been identified as part of a pair that meet the P8 rules for instruction fusion.

  `f` This instruction has been identified as a potential fusion candidate. The other instruction in the pair might be far away, and the instruction types and/or register and constant usage do not conform with the P8 rules.

- Next you see the instruction disassembly field
- After the disassembly field are the annotations:

  `L:nnnn` This is line nnnn of the source file of the current function.

  `D:nn` The *discriminator* number for this instruction at this line as given by the symbol table information.

  `N:nnnn` This instruction was executed `nnnn` times in the trace.

  `HOT-blk` The instruction is in a block that made the "Top 10" list of most executed blocks.

  `HOT-LongLat` This block made the "Top 10" list of blocks with frequently executed long latency instructions.

  `HOT-def-use` This instruction made the "Top 10" list of instructions with few intervening instructions and many cycles to the first dependent instruction.

  `Use:n` The register result from this instruction is used `n` instructions later. If the input is scrollpipe, this can be followed by `nn.ncy` indicating the average number of cycles between the finish event of this instruction and the issue event of the first dependent instruction using the register result of this instruction.

  `LBAD nn` This instruction is marked as a bad long-latency instruction with weight `nn`.

`PLT` This instruction is in a block flagged as being a PLT stub generated by the linker.

`NOP` This instruction was identified as being a no-op of some kind, generally or/ori/xori that has no effect.

`prolog` This instruction was identified as being part of the function prolog.

`TOC` This instruction was identified as being part of a sequence that computes an address relative to the TOC.

`TOCref` This instruction is part of a sequence that loads/stores to something in the TOC.

`short branch across CL`
Identifies a bc+8 or bc+12 that crosses a cacheline boundary.

`BL` This is a branch-and-link instruction.

`BLR` This is a branch-to-link-register instruction.

`bc+8` This instruction and the one following it meet the conditions for a pairable bc+8.

`LoopHdr NNi MM iter PPi dyn`
This instruction is the first of a block identified by control flow graph analysis as a loop header block. A total of `NN` instructions are in blocks that are part of the loop construct, the loop executes `MM` iterations, and a total of `PP` instructions are executed by the loop construct. For single-block loops `PP` can be expected to be the product of `NN` and `MM` but it may be different if there is control flow in the loop.

`Loop backedge`
This instruction is a branch back to the loop header block from the bottom of the loop. In some cases this can be an ordinary instruction if the back edge is a fall through to the loop header, or it could be a branch instruction that is not taken to fall through to the loop header.

`[[short] misaligned [32B] [128B]] loop nni`
A single block loop of `nn` instructions starts here. If prefixed by `misaligned` and `32B` and/or `128B` then it has more 32B and/or 128B boundary crossings than needed for the number of instructions in the loop. If this is prefixed by `short` then the loop is 16 or fewer instructions as well.

`Hint:unlikely`
This branch had the hint bits set indicating it was unlikely to be taken.

`Hint:likely`
This branch had the hint bits set indicating it was likely to be taken.

`[HOT] BAD HINT`
This branch was hinted but 10% or more of the time it did not go the way the hint said. If prefixed with `HOT` then this made the "Top 10" list of bad hinted branches.

`HOT-LBE` The branch was on the "Top 10" list of most likely to be mispredicted branches based on the Linear Branch Entropy metrics.

`[HOT-]Mispred: nn.n%`
: (scrollpipe only) This branch was mispredicted `nn.n%` of the time. If prefixed with `HOT-` then this made the "Top 10" list of mispredicted branches.

`from:0xXXXX`
: control flow branched to this instruction from address 0xXXXX. Up to 8 addresses will be listed. If there are more than 8, it will just say `from: nnn sources` and you can see the whole list in the per-instruction-address section that is enabled by command line option `-v`.

`to:0xXXXX`
: control flow branched to address 0xXXXX from here. Up to 8 addresses may be listed. If there are more than 8, it will just say `to: nnn targets` and you can see the whole list in the per-instruction-address section that is enabled by command line option `-v`.

`[HOT-LHS] DA ref [from|to] [intra|inter|inter-NN.N%] 0xXXXXX MMi P.PP% [QQi min] [CC.Ccy] [param RR] [disjoint 0xYYYYY SSi T.TT%] [LARX-LOOP] [larx-hit-larx LLx IIi JJ.JJ%] [larx-larx AAx BBi DD.DD%] [intra+stack: EE][inter+stack: FF][intra: GG][inter: HH]`
: indicates there is a data reference link between a load and a store.

  `HOT-LHS`
  : If prefixed by `HOT-LHS` then this one made the "Top 10" list.

  `from`
  : If this instruction is a load, it will say `from` and the addresses listed are the addresses of the stores that write data loaded here.

  `to`
  : For stores, it will say `to` and give the addresses of loads that use the data stored.

  `inter/intra`
  : If the execution path crossed a bl/blr, it will say `inter`, otherwise it says `intra`. If only some of the paths seen cross a bl/blr, then it will say `NN.N%-inter` meaning that `NN.N%` of the time the value is from a different function.

  `0xXXXXX MMi P.PP%`
  : After these it gives the address, number of instructions, and percent frequency of the most common case (in terms of the other instruction and the number of instructions between store and load).

  `QQi min`
  : If the minimum pathlength is less than the most common pathlength, then that is reported as `QQi min` meaning the minimum path is `QQ` instructions.

  `CC.Ccy`
  : If the input is an M1 scrollpipe file, the average cycle time between the finish of the source instruction and the issue of the destination instruction is `CC.C` cycles.

  `param RR`
  : `RR` instances of this store-load linkage were due to a compulsory load-hit-store on a stack-store function parameter.

  `disjoint 0xYYYYY SSi T.TT%`
  : The access at instruction address `0xYYYYY` is partially overlapping with the one at the current address. The minimum path length

between them is SS instructions and occurs T.TT percent of all store-load linkages involving this instruction.

LARX-LOOP This instruction is a larx that is in a loop that will get a larx-larx conflict between one iteration and the next.

larx-hit-larx LLx IIi JJ.JJ%
This instruction conflicts with another larx to the same address LL times with minimum pathlength II instructions which is JJ.JJ percent of all the larx-larx same-address conflicts for this instruction. If you ran pipestat with the -v flag, you can look for LARXH in the listing for the IA of this larx to find the IA of the larx that touched the address before.

larx-larx AAx BBi DD.DD%
This larx was preceded by another larx to any address within a window of 100 instructions. This window can be changed with command line option -badLarxLarx N. There were AA total occurrences, with minimum pathlength BB instructions accounting for DD.DD% of that total. As with larx-hit-larx, if you run with -v, you can look for LARXF in the listing for this larx to find the IA of the previous unrelated larx.

redundant: [intra+stack: EE][inter+stack: FF][intra: GG][inter: HH]
The data in destination register was never modified since the store that wrote the data now being loaded. The 4 counts distinguish data store on the stack (marked with stack) vs non-stack data, and intra-procedural (load and store in the same function context) vs inter-procedural (load and store are separated by a function call or return). Often this indicates that registers are being reload that were never modified due to a function doing an early exit.

larx-flush NN
(scrollpipe only) This indicates that NN pipeline events were seen for this instruction address that indicate a larx flush due to hitting a younger larx occurred.

cy:nn,nn,nn
(scrollpipe only) minimum, average, and maximum cycles seen for this instruction address to go from when it is first seen in the pipeline (fetch) to completion.

store flush NN
(scrollpipe only) This is the total number of LHS related flushes events seen for this instruction across all the times it was executed.

br pr:nn.nn% (avg seq tk x.x ntk y.y LBE g.ggg/l.lll)
The probability of actually branching if this instruction is a conditional branch. The branch probability is nn.nn percent, and on average the branch is taken x.x times in a row, and not taken y.y times in a row. The Linear Branch Entropy metrics for this branch are g.ggg global and l.lll local. If you have specified the -v option, the detailed output for the instruction will also have the number of taken/not-taken sequences and the standard deviation for the sequence lengths.

5.1 Code Score

The code score output is intended to help when you have a large number of traces to look at. This might be used to compare traces of the same source code compiled with different compiler options or different versions of the compiler. Or you can use it to look at all the traces in a large bank of different traces to help pick out ones with the characteristics you are interested in.

```
HDR: file, n_inst, n_reg_moves, n_redundant_loads, n_LHS_short, \
   n_misalign_short_loop_iter, branch, n_bad_hint, n_mispredict, \
   cyc_mispredict, LBEloc, LBEglob, num_basic_blocks, min_bb, \
   max_bb, avg_bb, exec_wgt_avg_bb, unique_IA, unique_DA
SCORE: perlprime.pipe, 118199, 0, 567, 5688, 0, 16267, 6, 230, 4328, \
   0.0059133295, 0.0157790927, 1061, 1, 62, 5.5466540999, 5.7525868016, \
   5885, 1207
```

With the `-score` flag, `pipestat` provides a single line of output with the fields given in the table below. The `-scorehdr` option outputs a header line with the field names as well as the code score line itself. If neither option is given, the score header and the score line are output at the end of the full `pipestat` output. The header line starts with `HDR:` and the score line starts with `SCORE:` so you can easily use `grep` to extract them from full `pipestat` output files.

`file` The trace file that was processed.

`n_inst` The number of instructions that were processed from the trace.

`n_reg_moves`
: The number of register-to-register moves seen. This is an interesting metric of compiler performance. If the compiler is doing a poor job of register allocation or is not managing multiple register classes (i.e. FPR/vector/VSX) well, there may be more register-to-register move instructions executed.

`n_redundant_loads`
: Redundant loads are cases where there is a store of some register to memory followed later by a load from that memory location back into the same register, with no intervening modifications of the memory location or the register. This can indicate that the compiler has inserted unnecessary spills of registers to memory, or that the *shrink wrap* optimization did not happen on a function with an early exit leading to extra registers being saved and restored unnecessarily.

`n_LHS_short`
: This is the dynamic number of load-hit-store dependencies that have less than 100 instructions between them (the `-badLHS` option sets this distance).

`n_misalign_short_loop_iter`
: This counts up all the iterations of short (fit in one cache line if optimally aligned) single-block loops that are not optimally aligned. These loops span more 32B or 128B crossing than would be necessary.

`branch` The number of branch instructions of all types. This is useful when comparing different traces to see which ones are the most branch-heavy.

`n_bad_hint`
: Branches are considered to be badly hinted if they don't go the way the hint says at least 90% of the time. This is the count of the number of times that branches flagged as badly hinted went in the opposite direction from the hint.

`n_mispredict`
: This is only nonzero if the input was a scrollpipe file. The number of branches that had "Branch flush, guessed wrong" events is summed up.

`cyc_mispredict`
: This is also only available for scrollpipe input. For each mispredicted branch, the penalty cycles are approximated by the difference between the finish cycle of the branch and the finish cycle of the next instruction after the branch. This is the sum of those penalty cycles across all mispredicted branches in the trace.

`LBEloc`
: This is the local version of Linear Branch Entropy, where previous decisions at that branch are used to index a table to look at future behavior. This score component is the average local LBE across all conditional branches in the trace.

`LBEglob`
: This is the global version of Linear Branch Entropy, where previous decisions at all conditional branches are used to index a table and look at future behavior of this branch. This score component is the average global LBE across all conditional branches in the trace.

`num_basic_blocks`
: The number of basic blocks identified by control flow analysis. Each block is a contiguous sequence of instructions with no branches in or out between the beginning and the end.

`min_bb`
: The minimum number of instructions in a basic block.

`max_bb`
: The maximum number of instructions in a basic block.

`avg_bb`
: The average number of instructions per basic block.

`exec_wgt_avg_bb`
: The average number of instructions per basic block, weighted by execution frequency.

`unique_IA`
: The number of unique instruction addresses.

`unique_DA`
: The number of unique data addresses referenced.