MCR10029
Fix PL/1 Compiler's Code Generation for Bitstring Concatenation

Eric Swenson

February 9, 2017

1 Issue

During Multics/emulator testing, Charles Anthony discovered that the PL/1 compiler was incorrectly compiling a bit string concatenation such as the following:

```
dcl  fbits bit (36);
dcl  tbits bit (52) aligned;

tbits = fbits || (16)"0"b;
```

When a non-zero initial value of fbits is used, the resulting value of tbits is (32)"0"b rather than fbits concatenated with (16)"0"b.

The following ticket describes this issue:

http://multics-trac.swenson.org/ticket/9

2 When Issue Was Introduced

The issue was not present in the version of the PL/1 compiler in MR12.3 (32f), but was introduced in a compiler update between MR12.3 and MR12.5. The version of the compiler in MR12.5 was 33e.

We traced the issue to a change made to the PL/1 compiler source `cat_op.pl1` in MR12.4, resulting in a compiler whose version number was 33a. While we do not have that compiler, the `cat_op.pl1` source was unchanged from version 33a to 33e.

Below is the history comment for the change to `cat_op.pl1` as it appears in `pl1_version.cds`:

```
11) change(1990-05-03,Huen), approve(1990-05-03,MCR8169),
    audit(1990-05-18,Gray), install(1990-05-30,MR12.4-1012):
    Updated version to 33a for pl1 opt concat of a common string exp bug
    (pl1_1885)
```

The history comment in `cat_op.pl1` is shown below:

```
 2) change(90-05-03,Huen), approve(90-05-03,MCR8169),
     audit(90-05-18,Gray),
    install(90-05-30,MR12.4-1012):
    pl1_1885: Fix pl1 optimizer to handle the concatenation of a common
        string
    expression correctly.
```

While this history comment suggests that it fixes a bug in the PL/1 optimizer, it turns out that, with the 33e compiler, incorrect code is generated regardless of whether the `-ot` control argument is provided to the compiler command line.

Unfortunately, we have no details on the ticket referenced in the history comment (pl1_1885).

3 Analysis

The code change made in 33a to `cat_op.pl1` is shown below:

```
6685c6688,6689
<       call compile_exp(q2);
---
>       if p2 -> reference.ref_count < 1 then call compile_exp(q2);
>       else p2 = compile_exp$save(q2); /* needed later */
```

Since before the fix, the code on the first branch of the above `if` statement would have been executed, we surmised that after the fix, the compiler must be taking the second branch of the `if` statement when bad code is generated and that the call to compile_exp$save(q2) was somehow causing the bug.

If we examine the code generated by the compiler in version 32f, we see this code:

```
tbits = fbits || (16)"0"b;
```

```
000432 aa  2 00002 3735 20 epp7     pr2|2,*
000433 aa  003 100 060 500 csl      (pr),(pr),fill(0),bool(move)
000434 aa  7 00000 00 0044 descb    pr7|0,36          fbits
000435 aa  6 00056 00 0044 descb    pr6|46,36
000436 aa  6 00056 2351 00 lda      pr6|46
000437 aa   000000 2360 07 ldq      0,dl
000440 aa  6 00146 7571 00 staq     pr6|102           tbits
```

The corresponding code generated in 33e is:

```
 tbits = fbits || (16)"0"b;

000432 aa  2 00002 3735 20 epp7     pr2|2,*
000433 aa  003 100 060 500 csl      (pr),(pr),fill(0),bool(move)
000434 aa  7 00000 00 0044 descb    pr7|0,36          fbits
000435 aa  6 00260 00 0044 descb    pr6|176,36        fbits
000436 aa   000000 2360 07 ldq      0,dl
000437 aa  6 00146 7571 00 staq     pr6|102           tbits
```

As can be seen, the `a` register is never loaded with the target of the `csl` instruction.

Just after the call to either compile_exp(q2) or to the call to compile_exp$save(q2) is a `goto` statement to the label `aa_2a`.

Looking at the code near this label, we see the following (in 33e):

```
aa_2:    if size2 = bits_per_word & mod(code,2) ^= 0
         then do;
              call expmac((lda),p2);
aa_2a:        call expmac((ldq),p3);
              end;
         else do;
aa_3:         call compile_exp(q3);
```

As can be seen, the `aa_2a` label comes just after the code that generates the required `lda` instruction.

It turns out, however, that simply moving the label to the previous instruction (the code that generates the `lda` instruction) is not the correct fix. This only works for one of the two branches of the above-mentioned `if` statement.

The correct fix is to branch to the `lda` generating instruction only in the second branch of the `if` statement and to the `ldq` generating instruction in the first branch.

Making this change does cause the correct code to be generated and fixes the bug.

4 Gaining Confidence

Because we lack compiler-writing skills, and because making a change such as that described above could have a negative impact somewhere else, we modified the 33e compiler to emit a message whenever the compiler exercised the code in either branch of the code, and we compiled the entire system (hardcore, sss, tools, and unb). The following PL/1 programs emitted a message indicating that they might potentially be impacted by any change to this code:

hardcore: init_empty_root.pl1, tape_mult_parse_.pl1, log_wakeup_.pl1

sss: check_info_segs.pl1

tools: tolts_ttyio_end_pl1, mtdsim_.pl1

unbundled: comp_write_page_.pl1, process_compout.pl1, **ibm2780_.pl1**, gtss_abs_.pl1, apl_parse_.pl1

The only file that was actually compiled with the 33e compiler was `ibm2780_.pl1`. So it made sense to investigate whether bad code was generated when this segment was compiled, or whether compilation takes the first branch of the `if` statement.

There were no messages emitted when we recompiled the compiler, indicating that the code path in question was not exercised when compiling the compiler.

Because the fix involves only the second branch of the `if` statement, we thought it would be interesting to see which branch the compiler takes when compiling each of these programs. We modified the compiler to incorporate the proposed fix (jump to the `lda` instruction in the second branch) and to print a message indicating which branch was being taken and then recompiled all these modules. Here are the results:

init_empty_root: branch 2. Code is good. LDA is correct.

tape_mult_parse_: branch 1. Unaffected by new fix.

log_wakeup_: branch 1. Unaffected by new fix.

check_info_segs: branch 2. Code is good. LDA is correct.

tolts_ttyio_end_: branch 1. Unaffected by new fix.

mtdsim_: branch 1. Unaffected by new fix.

comp_write_page_: branch 1. Unaffected by new fix.

process_compout: branch 1. Unaffected by new fix.

ibm2780_: branch 1. Unaffected by new fix.

gtss_abs_: branch 1. Unaffected by new fix.

apl_parse_: branch 1. Unaffected by new fix.

Inspection of the only two modules that exercised the second branch (init_empty_root and check_info_segs) shows that the code is correct and compares well with the same code compiled with the 32f version of the compiler. The `lda` instruction is present and does load the value put there by the `csl` instruction.

4.1 Unoptimized Compilers

To make sure that the proposed change has no negative impact when code is compiled without the `-ot` control argument (i.e. not optimized), we compiled all the segments that we identified as exercising the code path inquestion without the `-ot` option. We compared the code generated between the objects compiled by the 32f compiler and those generated by the new 33f compiler. There were no significant differences.

5 The Fix

The proposed fix is shown below:

```
cpa [lpn cat_op.pl1] cat_op.pl1

A208                    call expmac((lda),p2);
A209     aa_2a:         call expmac((ldq),p3);
Changed by B to:
B208     aa_2a:         call expmac((lda),p2);
B209     aa_2b:         call expmac((ldq),p3);


A257                    if p2 -> reference.ref_count < 1 then call
    compile_exp(q2);
A258                    else p2 = compile_exp$save(q2); /* needed later */
A259                    goto aa_2a;
Changed by B to:
B257                    if p2 -> reference.ref_count < 1 then do;
B258                         call compile_exp(q2);
B259                         goto aa_2b;
B260                         end;
B261                    else do;
B262                         p2 = compile_exp$save(q2); /* needed later */
B263                         goto aa_2a;
B264                         end;

Comparison finished: 2 differences, 15 lines.
r 21:27 1.123 19
```

As can be seen, the second branch results in a jump to the `lda` loading code, whereas the first branch jumps to the same place it always did – the `ldq` loading code.

It is safe to move the `aa_2a` label since the only place where this label is jumped to is where it appears in the update version of the compiler – from the second branch of the `if` statement.

6 Bootstrapping the Compiler

According to those who remember how things were done in the "old days", whenever the compiler was changed, it was always recompiled and bootstrapped – at least this is what was done before development of Multics moved to ACTC. It appears that the compiler was not recompiled for the MR12.3, MR12.4, and MR12.5 releases.

This MCR proposes that we once again bootstrap the compiler with the above fix. This ensures any changes do not break the compiler's ability to compile itself.

The bootstrap process is described below, courtesy of Monte Davidoff:

1. Use the currently released pl1 to compile the source of the new version of the compiler into a directory, say `b1`.
2. Use `b1>pl1` to compile the source of the new version of the compiler into a different directory, say `b2`.
3. Use `b2>pl1` to compile the source of the new version of the compiler into a different directory, say `b3`.
4. Using `compare_object`, compare `b2>pl1` and `b3>pl1`. If they are the same, release `b3>pl1` as the next version of the compiler. If they are not the same, something is broken: do not release the compiler.