# Fixing Register Coloring in VPO to the Power Architecture

A. J. Carpio, C. G. Rawls, B. F. Rayfield
North Carolina State University
Raleigh, NC USA

## Abstract

The Very Portable Optimizer (VPO) is an optimization framework that implements language- and architecture-independent compiler optimizations. The *Condor* team worked on the recently-added Power port to VPO, concentrating on problems with the global register coloring algorithm. Our team located two problems within the allocator that resulted in incorrect code. The first dealt with using scratch registers across function calls. The second was a bug in the reallocation of function parameter registers. Our team solved the first problem and found a temporary solution to the second problem.

## Introduction

The Very Portable Optimizer (VPO) is an optimization framework created at the University of Virginia. All optimizations implemented using VPO operate at the Register Transfer List (RTL) level. Since all optimizations operate at this level, the optimizer is language independent.

VPO is currently being ported to the Power Architecture from IBM. Therefore, not all optimizations are implemented or working. In this report we present the results of our work in fixing register allocation on the Power architecture.

## First Round Testing

Our initial tests focused on the *ctest* directory. We determined the similarities in the errors by running the *testit* script and observing the output. We immediately saw that one common bug was that multiple calls to printf() resulted in zeros being output on calls after that first.

This started us down the road of fixing the registers being allocated. We chose to then focus on the test: *test007.c*. This test was chosen because it was the simplest to exhibit the errors common to many tests. Below is the source code:

```
main()
{
    unsigned char c;

    c = 132;
    printf("%c [v]\n", (unsigned char) 'v');
    printf("%d [132]\n", c);
}
```

Within the *ctest* folder, all tests print a value along with the correct value in brackets. A test had run correctly when the first outputted value matches the value in brackets. In the initial register coloring implementation, *test007* outputs the first character correctly. However, the output of the second printf() is a zero instead of the value 132. The next step in our work was to examine the assembly output for this test, printed here: (author comments in **bold**)

```
#source statement 5
#Value 132 assigned to register 8
        li      8,132
#source statement 6
        li      4,118
        li      5,255
#printf() function loaded
        lwz     6,L0_TC_LBL_FUNC_printf(2)
        lwz     3,L0_TC_LBL_15(2)
        stw     4,56(1)
        lwz     7,56(1)
        and     4,5,7
        stw     2,20(1)
        lwz     5,0(6)
        lwz     2,4(6)
        mtlr    5
#printf() function called
        bclrl   20,0
        lwz     2,20(1)
#source statement 7
#Value register 8 moved to register 4
        mr      4,8
        li      5,255
#printf() function loaded
```

```
        lwz     6,L0_TC_LBL_FUNC_printf(2)
        lwz     3,L0_TC_LBL_16(2)
#Register 4 passed to second printf()
        stw     4,56(1)
        lwz     7,56(1)
        and     4,5,7
        stw     2,20(1)
        lwz     5,0(6)
        lwz     2,4(6)
        lwz     11,8(6)
        mtlr    5
#printf() function called
        bclrl   20,0
        lwz     2,20(1)
```

Investigation revealed that the character was initially allocated to register 8. Next, the first printf() function is called and parameters are passed to it. Then the second function is called, and the value in register 8 is passed in. After crossing the first function call, however, the value in register 8 has been changed and is now 0, so *test007* fails.

The Power architecture has volatile (caller-save) and non-volatile (callee-save) registers [1]. Volatile registers are sometimes referred to as "scratch" registers. When control is passed to another function, all values in volatile registers must be assumed to have been changed by the called function. If a value in a volatile register must be maintained across a function call, it must be saved before the call and restored afterwards. A more efficient strategy is to find all values that must be maintained across a call and place them in non-volatile registers, so that the caller does not need to save and restore them.

The cancolor() function in *assign.c* is used by the register coloring algorithm to find an appropriate register for a specific live range or blob of life. The type of register needed is passed in, along with the *ifcheap* variable. *ifcheap* is true if a register is used infrequently and is a candidate for assignment to a scratch register. Currently, some registers are incorrectly assigned to scratch registers that are live across a function call. In our modification to cancolor(), these registers are now assigned to non-volatile registers, because the *ifcheap* flag is used to determine which type of register to use.

## Second Round Testing

After fixing the local variables being allocated to scratch registers even though they were labeled not cheap, all of the tests in *ctest* worked. However, when we expanded our testing to the other directories, we found that the *misc* directory had several tests that failed to compile.

Further inspection required analyzing the assembly output of a test in *misc*. Word count, *wc.c,* seemed like the simplest to follow in assembly form. Stepping through the debugger line by line with the assembly we came across the problem. Whenever the function wcp() was called, it took four arguments. However, Those four arguments were not saved to non-scratch registers. The assembly code was trying to load the arguments from the registers that they were passed to the function, but the values were used across function boundaries, resulting in the same output problems that were fixed earlier. This was causing a zero to always be printed in the second column and a negative one in the third column while the correct value was printed in the first column.

Having this knowledge in hand, we proceeded to view debug output to determine what it was doing. The graph-coloring algorithm had allocated a non-scratch register to each parameter. However, another function had changed the function after this optimization. Inserting dump_function() statements into VPO allowed us to determine that fixentry() was altering the function.

Then, we inserted dump_function() calls and printf()s. We then found the section that derives the parameters from the registers and implements a renaming mechanism. This mechanism was renaming the allocated non-scratch registers to the scratch registers.

This mechanism is similar to the other architectures so we compared it to the MIPS architecture, one very similar to the Power. This didn't immediately yield any clues. The decision to rename though was based upon the *isused* flag of each register. So our attention turned to this variable. Currently, the values are not correct. This is where we currently stand. If we insert the

following code at the beginning of *fixentry()* in regs.c:

```
for (rnum = 0; rnum < MAXREGS; rnum++)
        isused[rnum] = TRUE;
```

most of the tests that did not work before begin to work. The word count program works as now the parameters are stored in non-scratch registers. However, the problem still exists with the sum of all totals when it performs word count on multiple files. This shows that either our temporary solution causes other problems with the register allocator or there are further VPO problems that we have not yet pinpointed.

Since that solution was a simple hack we needed to find out where this variable was set and where we should then set it ourselves. Before we started our project, only ***local_register_assignment***() located in assign.c altered this variable. This indicated that any allocations made during coloring OR linear allocation were not being noted for future use in fixentry.

To fix this we added three lines of code to locals.c that set the appropriate element of ***isused*** to TRUE when ***cancolor*** or ***canreplace*** returns true.

```
if (cancolor(type, allocate, reg, ifcheap)) {
        /*
         *  Let fixentry know that this register
         *  is used in this function.
         */
        isused[regtoindex(type, regnum(reg))] =
TRUE;
```

This allowed us to remove the hack that fixed register renaming. This also fixed many of the problems that the global linear allocator was running into.

## Remaining Issues

There were bound to be issues that could not be resolved. Below is a table listing most of the remaining issues. We first focused on ctest because those tests contained no conditional branches and were thus easier to debug. Once the compiler was able to build working versions of these programs we expanded our testing to other directories.

| Directory | Coloring | Linear |
|---|---|---|
| *ctest* | All tests work | All tests work |
| *avdhoot* | All tests work | All tests work |
| *cache* | error in compiled program. | error in compiled program. |
| *cq* | S626.c has error message: Vpo (s_add): unallocated item ‘r[158]’ | Output differs from gcc. Otherwise, this test works. |
| *espresso* | Error in compiled test. | Error in compiled test. |
| *jpeg* | Error in compiled program. | Error in compiled program. |
| *misc* | Sieve has errors. Related to allocation of an array on the stack. It forces the scalar variables to be placed at offset 32820 and beyond. Fm-part fails to run. | Same. An infinite loop is introduced in diag05.c |
| *msim* | All tests work | All tests work |
| *others* | All tests work | All tests work |

## Conclusion

VPO is designed to be a portable optimization framework, allowing a programmer to develop a port for a specific machine architecture and immediately benefit from all the optimizations already implemented by VPO. However, it is not always easy to create a full port to VPO, and problems can arise. Our team investigated problems with the register-coloring algorithm in the Power port to VPO. The team found two separate issues with the register allocator that was producing incorrect code. At the end of our project, the team has: a complete fix for the first problem of allocating volatile registers where non-volatile registers are needed, and a temporary fix to the second problem of incorrectly reallocated parameter registers. The fixes allow a high percentage of the

*powerpc-vpo* test cases to run correctly. When the temporary fix is finalized, it will provide a much more stable global register-coloring algorithm for the Power port to VPO.

## References

1. “PowerPC Assembly.” IBM developerWorks. <http://www-106.ibm.com/developerworks/library/l-ppc/>

## Website

http://www4.ncsu.edu/~ajcarpio/791A/condor.htm