Debugging a Local Register Allocator

COMP 412, Rice University
A Work In Progress

1 Introduction

The first programming assignment in COMP 412 proves challenging for many students. The assignment, or “lab,” requires students to write a local register allocator for a ten-operation subset of the ILOC intermediate representation. (See the lab handout and Appendix 1 of EaC2e.) Students must build a scanner, a parser, create an internal representation of the input program, and manipulate that program in complex ways. The lab requires students to track lots of detail. Some students find themselves unprepared to debug their register allocators in a time-efficient and effective way.

This brief document suggests a number of strategies that have helped COMP 412 students in previous years. It collects together, in a single place, advice that may help a COMP 412 student attack the problem of producing a working allocator. Many of these strategies apply to the third programming assignment in COMP 412 as well.

2 Understand what your code is doing

To gain an understanding of how the code in your allocator actually affects the compiled code, compare the execution traces of the original code and your allocated code. Run the original code in the ILOC simulator with the trace option (\texttt{-t}). Be sure to specify any command-line inputs that are needed; these are documented in the \texttt{SIM INPUT} line in the ILOC file’s header. Run the code produced by your allocator in the ILOC simulator with the trace option, as well.

The load, store, and loadI operations will likely be different between the two traces, as will the register numbers. However, the values that appear as inputs to the arithmetic operations (add, sub, mult, lshift, and rshift) should be the same. Comparing these values will lead you to the point where the computation begins to go awry.

Consider block \texttt{cc3.i} from Code Check 2. Figure 1 shows the execution trace of the original code. Figure 2 show the execution trace of the code after allocation by a student’s incorrect allocator. For the sake of brevity, we have elided cycles 11 through 36 in Figure 2.

In both traces, the first add, in cycle 1 produces a result of 0. So far, so good.

Now examine the second add, at cycle 4 in both traces. In the original, it has two distinct input registers (\texttt{r0} and \texttt{r1}) and produces the value 2. In the allocated code, it has two uses of the same register (\texttt{r2}) and produces the value 0. Comparing the traces immediately suggests that the allocator has the bug described in Section 4.1.

This bug occurred so early in the trace that no spill code had yet been generated, so the cycle numbers in the original and allocated code were identical. Farther along in the trace, the cycle numbers will not match. For example, the stores at cycles 29 and 30 in the trace of the original code, correspond to the stores at cycles 37 and 42 in the allocated code. Thus, the code at cycles 38 through 41 was inserted by the allocator; it performs a restore of the value used as an address in the last store.

You can use the arithmetic operations as guideposts to match equivalent lines in the two traces. The allocator should not add arithmetic operations, so by counting those operations, you can build a simple map from the original code, which is presumed correct, into the allocated code. Once you
locate the first operation where the operands or the result differ, it should point the way to the problem in your allocator.
3 Program Defensively

Design the allocator for debugging. Specifically, include output statements that print, to the standard error stream, enough information to allow you to track its actions and decisions. You can remove these statements later, but assume, up front, that you will need them. Similarly, add code to your allocator that explicitly checks the integrity of its key data structures. Again, once the allocator is working, you can remove these checks (or turn them into comments).

Check The Maps For critical data structures, build in consistency checks. A typical local allocator maintains a couple of mappings, such as PR_TO_VR and VR_TO_PR. Both of these maps should be one-to-one mappings (isomorphisms). A common source of problems in debugging the local register allocator lab is a failure to update these maps correctly. If either of them becomes an onto mapping—that is, multiple virtual registers map onto a single physical register or vice versa—then the allocator will undoubtedly generate incorrect code. Similarly, if they become inconsistent—the PR_TO_VR map shows values in registers that are not in the VR_TO_PR map, for

---

1 We use the term “map” in its generic, mathematical sense. This discussion is not intended to suggest that they should be implemented with a map data structure.
example—then the allocator is likely to lose the use of some registers.

Pick a point at your code where these two maps should be correct—for example, after you have completed all processing on a single operation. Add code that checks, explicitly, whether or not the \texttt{PR\_TO\_VR} and \texttt{VR\_TO\_PR} maps are correct. Check the condition:
\[
\forall \text{ physical register } p, \quad \texttt{VR\_TO\_PR} [\texttt{PR\_TO\_VR} [p]] = p
\]
Similarly, check the condition:
\[
\forall \text{ virtual register } v, \quad \text{if } \texttt{VR\_TO\_PR} [v] \text{ is defined, } \texttt{PR\_TO\_VR}[\texttt{VR\_TO\_PR}[v]] = v
\]
If either condition is false, then one or more updates to these maps is wrong, and the allocator will, almost certainly, produce incorrect code.

**Check For Consistent States** As it finishes with each operation, the allocator should be in a consistent and correct state. That state can be verified in multiple ways.

For example, at the end of each operation, the next-use information for each value in a physical register should be consistent with the next-use information at the most recent appearance of that register in the intermediate representation. As it runs through the code performing allocation and assignment, the local allocator needs to track the next use of each value that currently occupies a physical register. If the updates to the physical-register next-use information are incorrect, that can produce a variety of bugs in the allocated code.

Build in a check that ensures that the next-use data makes sense. For example, after the allocator finishes with operation \(i\), every value stored in an allocated physical register\(^2\) should have a next use in an operation whose index is larger than \(i\). That value should match the next-use value at the most recent appearance of the corresponding virtual register in the IR.

**Lab 3: Check The Dependence Graph** In Lab 3, you will build an instruction scheduler. The critical data structure in that project is a graph. At the end of the graph-building phase (and, if your lab has one, the graph simplifier), it should check the structure of the graph. If an edge believes that it connects node 1 to node 17, the nodes should believe that the edge exists and runs from node 1 to node 17.

4 Bugs Specific To The Local Allocator

4.1 Two Uses, One Register

One common bug occurs when the allocator fails to recognize that it needs two distinct registers for two distinct uses. Consider the \texttt{add} operation shown in the \texttt{Before Allocation} column.

\[
\begin{array}{c|c}
\text{Before Allocation} & \text{After Allocation} \\
\hline
\texttt{add vr2, vr3} & \texttt{add r0, r1} \\
& \Rightarrow \texttt{vr17} \quad \Rightarrow \texttt{r1} \quad \textit{could be correct} \\
\texttt{add r0, r0} & \texttt{add r0, r0} \\
& \Rightarrow \texttt{r1} \quad \textit{cannot be correct}
\end{array}
\]

The first operation in the \texttt{After Allocation} column is, potentially, a correctly allocated version of the pre-allocation \texttt{add}. \texttt{vr2} has been allocated to physical register \texttt{r0}; \texttt{vr3} has been allocated to

\(^2\)As opposed to a \texttt{free} or \texttt{unoccupied} physical register
physical register \( r_1 \); and \( vr_17 \) has been allocated to \( r_0 \). Assuming the right code before and after this version of the \texttt{add} operation, this allocation can be correct.

In contrast, the second example operation in the After Allocation column cannot be correct, unless \( vr_2 \) and \( vr_3 \) are the same live range. (In that case, they would have the same \( vr \) number!) The most likely source of this common bug is that the allocator did not mark \( r_0 \) as in use for the operation and, therefore, unavailable for allocation when looking for a physical register for \( vr_3 \).

<table>
<thead>
<tr>
<th>Before Allocation</th>
<th>After Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{store vr1} ( \Rightarrow ) ( vr_2 )</td>
<td>\texttt{store r0} ( \Rightarrow ) ( r_1 ) \textit{could be correct}</td>
</tr>
<tr>
<td>\texttt{store r0} ( \Rightarrow ) ( r_0 )</td>
<td>\texttt{store r0} ( \Rightarrow ) ( r_0 ) \textit{cannot be correct}</td>
</tr>
</tbody>
</table>

As a general issue, the code that looks for a physical register needs to understand that it cannot use the same physical register for two distinct virtual registers in a single operation. Typically, the code that allocates and assigns a physical register to the first operand must also mark that physical register as unavailable for the second use.

A special case of this bug arises with the \texttt{store} operation. The virtual register to the right of the arrow in a store operation is a use, rather than a definition. Thus, if the two virtual registers in a \texttt{store} operation have different virtual register numbers, they must be assigned different physical registers.

5 Performance Debugging

Before spending a lot of time on performance, you should figure out whether or not you have a problem. As a first step, run your code in the timing harness and compare the timing curve for your implementation with the timing curve shown in the back of the lab handout for your implementation language. If your lab is within 10% of the best time, it is probably fast enough. (You can always try to beat the best time.) If your lab is much slower than the best time—say a factor of 1.5 or more slower—then you should look at improving its efficiency.

The first step in any performance debugging effort is to discover where your program is spending its time. This may be easy or it may be difficult; that depends on the tools available for your implementation language. If you have easy access to a profiler, use it. If you have no tools at all, you can measure the time at various points in the code and print it out. (On \texttt{CLEAR}, you can find the \texttt{c} code for a POSIX-compliant microsecond timer in the directory \texttt{students/lab1/debug} of \texttt{comp412}'s account.)

Here are some specific problems that students have had in the past.

- \textit{String comparisons} Some students keep their internal representation in string form. Comparing two strings is expensive relative to comparing two integers. We recommend converting strings, such as the opcodes in ILOC operations or the names of registers, to small integers in your scanner or parser.

- \textit{String concatenation} Some students, recognizing that \texttt{i/o} operations can be expensive, have tried to concatenate all of the output operations into a single large string, which can then be printed in a single \texttt{i/o} operation. String concatenation is expensive (see §7.6 in EaC2e).
Expensive operations in Library Data Structures Many students have used Java’s ArrayLists or python NumPy arrays. Both of these data structures offer array-like features, coupled with expandability. Apparently, they work reasonably on small problems. On the larger timing blocks, however, they introduce serious performance problems when the allocator tries to insert or delete operations.

It is important to keep the cost of inserting and deleting operations as low as possible—specifically, those operations should require \( O(1) \) time with low, constant overhead. We would advise you to spend the time and build a real linked list that has \( O(1) \) cost to add to either the head of the list or the tail of the list; and \( O(1) \) to add at before an arbitrary list element, once you have the pointer to that element. (That probably implies a doubly-linked list with external points to the first and last elements of the list.)

Updating versus Printing If your lab traverses the block in order from top to bottom, you may be able to design it so that it simply prints the results, rather than updating the IR and then printing from the IR. In many cases, this approach produces a faster-running pass.

In Java labs, many students ignore the cost of managed storage—allocation and garbage collection. Excessive time spent in the garbage collector is a symptom of poor memory use. To test whether the code is collecting too much, try running your lab with a much larger heap (see the notes on Garbage Collection the comp412 website’s Lectures page). If excess collection is a problem, look for places where the code reallocates rather than reuses. For example, we have seen many student labs that create a new object at the start of each iteration of a loop, rather than allocating it outside the loop and re-initializing it (by assigning its fields known values) at the start of each iteration. An assignment is much cheaper than an allocation and very much cheaper than a collection cycle.