Auditing Register Allocation in Linear Time

Ralph E. Johnson
Donald Roberts

Register allocation is a well-known and important NP-complete problem [Set75]. NP-complete problems can be characterized as problems that are hard to solve but whose answers are easy to check. The fact that register allocation is an NP-complete problem has influenced the design of register allocation algorithms, but it also implies that there is a polynomial algorithm to audit (check the correctness of) the results of allocating registers. This paper presents the linear-time algorithm for auditing register allocation that is used in the TS compiler [JGZ88].

Modern register allocation algorithms use the fact that register allocation is NP-complete by reducing it to the graph coloring problem [CAC*81][CH84]. These algorithms have several components. They first determine the lifetime of each variable and build a conflict graph with edges between nodes representing overlapping lifetimes. They then color the graph so that no two adjacent nodes (variables with overlapping lifetimes) have the same color (register) using an approximation for the graph coloring problem. Sometimes registers have to be spilled to memory; there are many different algorithms for selecting registers to spill. In our experience, the graph coloring approximation is easy to implement correctly because it is simple and well-defined, while the other two components (building the conflict graph and spilling) are more difficult.

Auditing register allocation makes it easier to debug a compiler. Register allocation bugs are hard to find by inspecting the output of a compiler, because one must compare widely separated parts of the program to find them. Moreover, a slight change in the input of a register allocation algorithm can lead to completely different output. This is because converting a program into a conflict graph involves a complete change of data structures, and the

Authors' Address: Department of Computer Science, 1304 W. Springfield, Urbana, IL 61801 USA. *Email*: johnson@cs.uiuc.edu, droberts@cs.uiuc.edu *Phone*: (217) 244-0093

mapping from the domain of programs to the domain of conflict graphs is not easy for humans to follow. The order that physical registers are assigned to variables depends more on the structure of the conflict graph than of the program, so a small change in the input program may lead to a large change in the way registers are allocated.

Our algorithm for auditing the register allocator is an end-to-end test; *i.e.,* it does not depend on any of the intermediate results of the register allocation algorithm. Thus, it does not depend on the variable conflict graph or even the variable lifetime calculation, and so cannot be affected by bugs in them. The register allocation auditor does not even depend on the algorithm used to allocate registers. It depends only on the original program and the mapping from variables to registers produced by register allocation. Hence, the auditing algorithm provides an independent check of the register allocator.

Programs

A program is represented by a labeled directed graph whose nodes represent statements. The edges of the graph represent control flow. Edge labels represent conditions on control flow and are ignored by register allocation algorithms and the auditing algorithm.

A statement reads a set of variables and writes a set of variables. Of course, the values written to variables are a function of the values that are read, but register allocation algorithms ignore the functions being computed and focus only on the variables being read or written. Thus, the register auditor will examine the set of variables read and written by each statement, but will otherwise ignore the statement.

A program is defined as a set of statements S, a set of variables V, and three functions defined on S:

- $pred(s)$ is the set of predecessors of s ($pred : S \rightarrow 2^S$)
- $read(s)$ is the set of variables read by s ($read : S \rightarrow 2^V$)
- $write(s)$ is the set of variables written by s ($write : S \rightarrow 2^V$).

Although this representation does not capture all the information about a program, it is sufficient for many purposes. For example, it is sufficient for

interval analysis. An *interval* is a single-entry, strongly connected region. If a program graph has been put in reducible form then all loops are intervals [ASU86]. Many optimizing compilers put programs in reducible form and find intervals; we assume that this has been done before register allocation. An interval's single entry point is called the *interval head*. All back-edges in a program in reducible form are to an interval head. Nested loops result in nested intervals. The auditor will analyze the program one interval at a time. For each statement s, $head(s)$ is the head of the most deeply nested interval containing s. If s is the head of an interval then $head(s)$ is the head of the next most deeply nested interval; *i.e.* $head(s) \neq s$.

Register Allocation

Register allocation is more than computing a function that maps variables to the physical registers that are allocated to them. Since register allocation introduces register spills, it can change a program by adding copies between memory and registers. Because of register spills, register allocation computes a mapping from variables to a *set* of locations, both registers and memory, that store the value of a variable. We describe register allocation by augmenting the description of programs with a new set L, the set of locations, and with four functions:

- $isCopy(s)$ is a predicate that tells whether statement s was introduced during register allocation to spill a register.
- $loc(v, s)$ is the register or memory location used to store v for statement s, where s is not a copy. (The purpose of register allocation is to define this function.)
- $source(s)$ is the source location of copy statement s.
- $dest(s)$ is the destination location of copy statement s.

This program representation implies that non-copy statements use only one of a variable's locations. There is no reason to read from two different locations in one statement, but it might be useful to read from one location of a variable and write to another. Changing the program representation to allow this would not change the algorithms presented in this paper, but would complicate the notation.

Some machine instructions modify locations (usually registers) in addition to the ones representing variables that will store the results. For example, a procedure call will modify registers used by the procedures, and a division instruction might store the remainder in a register as well as storing the result of the division. The potential conflicts caused by statements that modify locations is described by adding variables to the statements' write sets that represent the modified locations but are not read by any statement.

The number of locations ($|L|$) is equal to the number of registers plus the number of variables that have been spilled. Usually few variables are spilled, but the worst case will be when all variables are spilled. The number of registers depends on the target machine, and so is a constant for any particular compiler-target machine pair. The number of variables is much smaller than the size of the program, so we will assume that $|L|$ is a constant. This is the same assumption made for flow analysis algorithms to achieve $O(n\ log\ n)$ times. The auditing algorithm that we will develop will be $O(n|L|)$, where n is the size of the program, so we will say that it is a linear-time algorithm.

Configurations

The register auditor is based on *configurations*, partial functions from locations to variables that indicate the variable whose value each location contains. Auditing a statement takes the configurations for the predecessors of a statement and computes the configuration for the statement. If *config* is a configuration for a statement s then $config(l) = v$ means that on every path through the program that ends at s, l contains the value that was last written to v. $config(l) = \perp$ means that l is not assigned to any variable. The functions $preconfig : S \rightarrow L \rightarrow V$ and $postconfig : S \rightarrow L \rightarrow V$ give the configuration before and after each statement, respectively. Register allocation is wrong if a statement s reads a variable v assigned location l, but $preconfig(s)(l) \neq v$.

Configurations have finite domains, the set of locations L. Thus, they can be treated as finite sets of pairs and their intersection can be computed. The definition of *preconfig* is

$$preconfig(s) \equiv \bigcap_{t \in pred(s)} postconfig(t).$$

This means that a location is in the domain of $preconfig(s)$ only if it is in

the domain of the postconfiguration of each of the predecessors of s and each maps it to the same variable.

The relationship between a statement's preconfiguration and postconfiguration depends on what kind of statement it is. A copy statement (introduced by a spill) only changes the configuration of the location that is written, while a noncopy statement not only changes the configuration of the location being written, but invalidates all other locations that store a copy of the variable. Location l is invalidated at statement s if s writes to the variable that l represents, but that variable is not assigned to l at s.

$$invalidated(l, s) \equiv \mathit{preconfig}(s)(l) = v \wedge v \in write(s) \wedge loc(v, s) \neq l.$$

Then

$$\mathit{postconfig}(s)(l) \equiv \begin{cases} \mathit{preconfig}(s)(source(s)) & \text{if } isCopy(s) \wedge l = dest(s) \\ v & \text{if } \neg isCopy(s) \wedge v \in write(s) \wedge loc(v, s) = l \\ \bot & \text{if } \neg isCopy(s) \wedge invalidated(l, s) \\ \mathit{preconfig}(s)(l) & \text{otherwise} \end{cases}$$

Given the function *preconfig* for a program, the program's register allocation is correct if and only if

$$\forall s \in S : \neg isCopy(s) \Rightarrow (\forall v \in read(s) : \mathit{preconfig}(s)(loc(v, s)) = v). \quad (1)$$

A Linear-time Algorithm

The definitions of *preconfig* and *postconfig* given above are dataflow equations, and can be used to define iterative or reduction-based dataflow algorithms, which typically have time bounds of $O(n^2)$ or $O(n \ log \ n)$ [RP86]. However, a little examination shows that the definition can also produce a linear-time algorithm.

Were it not for back-edges, it would be easy to calculate *preconfig* and *postconfig* in linear time. For any interval head s, *preconfig*(s) depends on *postconfig* for the predecessors of s, and *postconfig* for a predecessor of s on a back-edge depends on *preconfig*(s). Solving this recursive equation is what causes dataflow algorithms to take more than linear time.

The auditor computes an approximation to *preconfig* and *postconfig* in linear time by ignoring postconfigurations of predecessors along back-edges.

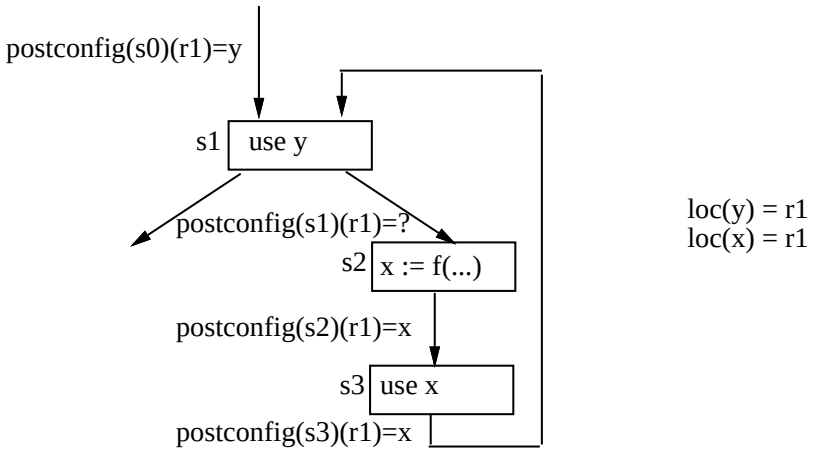

Figure 1: A Register Allocation Error

Ignoring a postconfiguration has a limited effect on computing a preconfiguration *config*: it might cause $config(l) = v$ to be computed for some variable v even though l is invalidated. Consider Figure 1. Since the interval head $s1$ uses y but does not write to x or y, a single pass algorithm that ignores the postconfigurations from back-edges when computing the preconfiguration of interval heads would compute $preconfig(s1)(r1) = y$. This would be wrong since the back-edge defines $postconfig(s3)(r1) = x$, so $preconfig(s1)(r1) = \perp$. All errors in calculating configurations in a single pass are of this form; the calculated configurations are defined for some locations for which they should be undefined. The auditor will compute an approximation *post* to *postconfig* (and *pre* to *preconfig*) such that $(\forall s \in S, v \in V, l \in L : postconfig(s)(l) = v \Rightarrow post(s)(l) = v)$.

After the auditor computes an approximation to *preconfig* and *postconfig*, it then (in linear time) finds any errors in the approximations and determines if they are significant. Errors in the approximations are always the result of ignoring back-edges. They might cause a register allocation error to be undetected, but they never mistakenly report an error. Errors in a preconfiguration are significant only if they are for locations that are read by a statement, since reading a location whose configuration is undefined implies an error in register allocation. For each interval head, the auditor

computes a set of locations read by statements in the interval. As long as the configurations along back-edges of an interval all agree for each location read in the interval, there are no missing register allocation errors.

Recall that $head(s)$ is the head of the most deeply nested interval containing s. $unchanged(s)$ is the set of locations l for which there is a path from $head(s)$ to s on which l is not written. Let

$$pred'(s) \equiv \{t \in pred(s) \wedge \overline{ts} \text{ not back-edge}\}.$$

Then

$$unchanged(s) \equiv \bigcup_{t \in pred'(s)} \begin{cases} L - \{loc(v,t) \mid v \in write(t)\} & \text{if } t = head(s) \\ unchanged(t) - \{loc(v,t) \mid v \in write(t)\} & \text{otherwise} \end{cases}$$

Instead of computing *preconfig* and *postconfig*, the auditor computes approximations *pre* and *post* such that *pre* has the property:

> **Main Invariant** $pre(s)(l) = v$ means that on every path through the program that ends at s without going through a back-edge to the head of one of the intervals containing s after a write to l, l contains the value that was last written to v.

The motivation behind the main invariant is to make it possible to compute $pre(s)$ in one pass over the program by ignoring loops that contain s. However, the main invariant implies that *pre* must not ignore other loops. This means that the postconfiguration of a statement depends on whether the statement's successor is in an enclosing interval, since the its postconfiguration in an enclosing interval must summarize back-edges, while its postconfiguration in the same interval does not. The function *exitpost* gives a postcondition for a statement that includes information about back-edges. A statement t's postconfiguration in an outer interval is given by

$$exitpost(t)(l) \equiv \begin{cases} post(t)(l) & \text{if } l \in unchanged(t) \Rightarrow \\ & \quad (\forall u \in S : \overline{u\ head(t)} \text{ a back-edge} \Rightarrow post(t)(l) = post(u)(l)) \\ \bot & \text{otherwise.} \end{cases}$$

Thus, $exitpost(t)(l)$ is the same as $post(t)(l)$ unless there is a path from t to $head(t)$ that does not write to l and there is a back-edge u to $head(t)$ such that $post(u)(l) \neq pre(head(t))(l)$.

A definition of pre that satisfies the main invariant is

$$pre(s) \equiv \bigcap_{t \in pred'(s)} \begin{cases} post(t) & \text{if } head(s) = t \vee head(s) = head(t) \\ exitpost(t) & \text{otherwise} \end{cases}$$

where

$$post(t) \equiv \begin{cases} pre(s)(source(s)) & \text{if } isCopy(s) \wedge l = dest(s) \\ v & \neg isCopy(s) \wedge v \in write(s) \wedge loc(v,s) = l \\ \bot & \neg isCopy(s) \wedge invalidated(l,s) \\ pre(s)(l) & \text{otherwise} \end{cases}$$

The main invariant is not strong enough to audit register allocation, since *all* paths should ensure that l contains the value that was last written to $pre(s)(l)$, not just paths that contain no back-edges to the head of any interval containing s. If a path contains a back-edge to the head of an interval containing s after the last write to l then let $\overline{tr}$ be the first such back-edge after that write. By the main invariant and the rule for calculating $post$, l contains $post(t)(l)$ at t. As long as $pre(r)(l) = post(t)(l)$ then by the main invariant, l contains the value last written to v, so the register allocation is correct for that path. However, it is not necessary that $pre(r)(l) = post(t)(l)$ for every back-edge $\overline{tr}$ and every label l. Often l is not used (is dead). It is only necessary for $pre(r)(l) = post(t)(l)$ if there is a statement s on a path from r that reads l without an intervening write to l.

Let $hasBeenRead(t) = \{l \in L : (\exists s \in \text{interval of } t, v \in read(s) : loc(v,s) \in unchanged(s) \wedge l = loc(v,s))\}$. This means that $hasBeenRead(r)$ is the set of locations at interval head r that are read before they are written. If $\overline{tr}$ is the first back-edge after the last write to l on a path to s and $l \in read(s)$ then $l \in hasBeenRead(r)$. Thus, l will have the last value written to v at s if $pre(s)(l) = v \wedge pre(r)(l) = post(t)(l)$.

In summary, register allocation is correct — that is, on every path through the program that ends at s, $v \in read(s)$ implies that $loc(v,s)$ contains the value of v — if and only if

$$\begin{array}{l} (\forall s \in S : \neg isCopy(s) \Rightarrow (\forall v \in read(s) : pre(s)(loc(v,s)) = v)) \wedge \\ (\forall s \in S : \neg isCopy(s) \wedge \\ \qquad s \in pred(head(s)) \Rightarrow (\forall l \in hasBeenRead(head(s)) : \\ \qquad\qquad post(s)(l) = pre(head(s))(l))) \end{array} \tag{2}$$

An algorithm to compute $hasBeenRead$ quickly is

```
for each s in topological order do
    if s is interval head then hasBeenRead(s) ← φ
    for each v ∈ read(s) ∧ v ∈ unchanged(s) do
        t ← head(s)
        while loc(v, s) ∉ hasBeenRead(t) do
            hasBeenRead(t) ← hasBeenRead(t) ∪ {loc(v, s)}
            t ← head(t)
```

For each interval head s, $hasBeenRead(s)$ is updated zero or one times for each location. Each statement will keep checking interval heads until it finds one for which $hasBeenRead$ does not have to be updated. Each statement will read $hasBeenRead$ without updating it zero or one times for each location. Thus, computing $hasBeenRead$ takes $O(|S| \times |L|)$ time.

All of these functions can be computed in one traversal of the program graph. The following algorithm combines all the functions described so far and tests whether register allocation is correct using equation (2).

for each s in topological order, finishing deeper intervals first, do

$$pre(s) \leftarrow \bigcap_{t \in pred'(s)} \begin{cases} post(t) & \text{if } head(s) = t \vee head(s) = head(t) \\ exitpost(t) & \text{otherwise} \end{cases}$$

$$post(s) \leftarrow \begin{cases} pre(s)(source(s)) & \text{if } isCopy(s) \wedge l = dest(s) \\ v & \neg isCopy(s) \wedge v \in write(s) \wedge loc(v, s) = l \\ \bot & \neg isCopy(s) \wedge invalidated(l, s) \\ pre(s)(l) & \text{otherwise} \end{cases}$$

$$unchanged(s) \leftarrow \bigcup_{t \in pred'(s)} \begin{cases} L - \{loc(v, t) | v \in write(t)\} & \text{if } t = head(s) \\ unchanged(t) - \{loc(v, t) | v \in write(t)\} & \text{otherwise} \end{cases}$$

if s is interval head then $hasBeenRead(s) \leftarrow \phi$
for each $v \in read(s) \wedge v \in unchanged(s)$ do
    $t \leftarrow head(s)$
    while $loc(v, s) \notin hasBeenRead(t)$ do
        $hasBeenRead(t) \leftarrow hasBeenRead(t) \cup \{loc(v, s)\}$
        $t \leftarrow head(t)$
if $\neg isCopy(s) \wedge (\exists v \in read(s) : pre(s)(loc(v, s)) \neq v)$ then **error**
if $\neg isCopy(s) \wedge s \in pred(head(s)) \Rightarrow$
    $(\forall l \in hasBeenRead(head(s)) : post(s)(l) = pre(head(s))(l))$ then **error**

The definition of *postconfig* has execution time at each statement $O(|L|)$. The definition of *preconfig* has execution time at each statement proportional to the number of in-edges and $|L|$. In structured programs, the cases of if and case statements can only be in one if or case, so either a statement has one out-edge or its successors have one in-edge. This implies that the total number of in-edges is never more than twice the number of statements. Thus, the amortized execution time of *preconfig* at each statement is $O(|L|)$.

Computing $hasBeenRead$ takes time proportional only to $|L|$ and the size of the program. Computing *post* and *unchanged* takes time proportional to the number of locations at each statement. Thus, the entire algorithm takes time $O(|L||S|)$. Given the usual assumption (for flow analysis algorithms) that $|L|$ is constant, the auditor is linear in the size of the program.

Conclusion

We have found the register allocation auditor to be a valuable part of the compiler. The auditor is fast and takes only a fraction of the time to allocate registers (usually around 20%), so there is little penalty for using it. It is able to detect every error made by the register allocator. When the compiler produces incorrect code, the auditor lets us concentrate on debugging parts of the compiler other than register allocation. This has proven to be very useful in building an optimizing compiler.

Acknowledgements

Sam Kamin, Mike Lake, and Carl McConnell gave valuable criticism on earlier drafts of this paper.

This research was supported in part by the National Science Foundation under grant CCR-8715752.

References

[ASU86] Alfred C. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[CAC*81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[CH84] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, pages 222–232, 1984.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):231–276, September 1986.

[Set75] Ravi Sethi. Complete register allocation problems. *SIAM J. of Computing*, 4(3):226–248, September 1975.