Finding free registers

Andrei Broder Monika Rauch Henzinger
Systems Research Center
Digital Equipment Corporation
Palo Alto, CA 94301
{broder,monika}@pa.dec.com

Abstract

We call a register r *free* at a node v of a program flow graph if every path in the flow graph that starts at v and ends at a read of r, contains a write to r. This note presents a series of algorithms for finding free registers, at one, at several, or at all points of a program. Their theoretical worst-case complexity does not improve over previous approaches, but experiments indicate that our algorithm are much faster in practice. The likely explanation is that for the standard approaches the worst-case is a common occurrence, while for our algorithms it arises only in pathological situations, not encountered in real programs.

1 Introduction

We call a register r *free* at a node v of a program flow graph if every path in the flow graph that starts at v and ends at a read of r, contains a write to r. Thus the value currently stored in r at v is immaterial for any possible execution path, since it is always overwritten by a new value before r is read. This note presents a series of algorithms for finding free registers.

We invented these algorithms for use in a tool that *patches* executable programs; that is, that replaces small portions of the code with new code, for debugging, performance instrumentation, architecture emulation, and similar purposes. The newly inserted code will be more efficient if it uses registers, but to preserve the semantics of the original program, the patch must use only registers that are available, that is, registers whose content at the point of interest has no further use by the original program under any execution path.

Our definition of "free" is purely syntactic. All free registers are available in the sense above but a semantic analysis of the program might find more registers that are available. We do not address this issue further, but note that in general the question whether a register is available is undecidable.

Determining which registers are free at various points in a program is a well-studied issue in compiler design, for a variety of reasons, including the determination of which registers must be saved and restored around procedure calls. Here however we are concerned with a somewhat different situation: we are given an executable, and we want to patch it at certain points. (The particular application that motivated us is the emulation of unimplemented instructions on older architectures.)

It is often the case that the space available for patches is limited, thus it is preferable to re-use patches whenever possible. This can be done only if the registers used by the patch are free at all locations where the patch is to be re-used. Hence there is some advantage to finding more free registers than strictly necessary, but there is a trade-off between the amount of resources (time and space) spent looking for free registers and the number of free registers that we can find. This is usually a non-issue if the looking is done at compile time, but often the patch is done dynamically in which case we have to be more careful. Depending on the application the user might be interested in free registers at one point, at several points within a particular procedure, or at all points within a program. The worst case behaviour of our algorithms for all these problems is the same, namely $O(E \cdot R)$ where E is the number of edges in the flow graph associated to the program, and R is the number of registers. In this sense, this is not an improvement over the previous approaches, see e.g. [1, 2], but experiments indicate that our algorithm are much faster in practice. This seems to be caused by the fact that the standard approach uses $\Omega(E \cdot R)$ time in all instances, while the worst cases for our algorithms arise only out of pathological situations, not likely to occur in a real program.

An obvious algorithm is to first construct the complete flow graph and then for each register, explore the edges of the graph backwards starting from all "reads" and stopping at all "writes", and marking the register as "in-use" for all the reached nodes.

However this could be very wasteful in practice – it might well be the case that only a very small portion of the flow graph is reachable from the point of interest. Furthermore, in a dynamic context, the user might want to limit the amount of time spent searching to a reasonable fraction of the expected gain. Thus it is preferable to have algorithms that can "time-out" graciously, that is the algorithm should be stoppable after a certain time and return some subset of the free registers. (The naive algorithm discussed above does not qualify since the time spent constructing the graph dwarfs the time spent searching.)

An obvious method for the discovery of free registers at a particular point is to do for each register a depth first search (DFS) on the portion of the flow graph accessible from the point of interest. The disadvantage of this method is that the flow graph has to be traversed many times and that we do not have the "gracious time-out" mentioned above. Our main contribution is an algorithm that does this series of DFS's simultaneously, thus finding quickly registers whose freedom is easy to deduce. In fact we have several algorithms, depending whether we are looking free registers at one point, at several points within a particular procedure, or at all points within a program. They all share the simultaneous DFS idea.

Our algorithms maintain for each node in the graph the set of trees known to contain it. During the execution of the algorithm, for each node this set can only grow, and at the end of the execution this set is exactly the complete set of trees containing the node, with the possible exceptions on registers known to be in use at the start point – see below. The efficiency of our algorithm comes from two facts: (a) in typical situations, whenever a node is visited for the first time this set is fairly close to its final value and there are only few increments afterwards (b) as soon as a register is determined to be in use, further exploration of the graph with respect to that register ceases.

2 Preliminaries

We associate to the program under consideration a directed graph. Every branch point or target of a branch is represented as a node, and edges represent the sequence of instructions between nodes. Often we have to deal with only a part of a larger program: in this case we might choose not to include branches outside the code of interest and thus there might be nodes, called *leaves* that have no outgoing edges. Leaves might also arise when the program includes branches whose target can not be determined. Of course, we make pessimal assumptions about the use of registers at leaves.

We define two predicates on edges and registers:

- $write(i, e)$ is true if there exists an instruction on edge e that uses register i and the first such instruction overwrites the content of i without using it.
- $read(i, e)$ is true if there exists an instruction on edge e that uses register i and the first such instruction uses the content of i.

An edge e that does not satisfy either $write(i, e)$ or $read(i, e)$ is said to be *clean with respect to register* i, and a path is said to be clean with respect to i if it is composed entirely of clean edges with respect to i.

We say that a register i is *in-use* at v if there exists a path from v that contains an edge e with $read(i, e)$ not preceded by any edge f with $write(i, f)$ or if there exists a clean path with respect to i from v to a leaf. Thus for each leaf v we make the (worst case) assumption that all registers are in use at v. A register not in use is called *free*.

Given these definitions, it can be easily seen that a register i is free at *Root* if and only if within every path from *Root* that contains an edge e with $read(i, e)$, the edge e is preceded by an edge f with $write(i, f)$, and there is no clean path with respect to i from root to leaf.

As usual, we define $succ(v)$, respectively $pred(v)$ to be the set of nodes $\{u\}$ such that there exists an edge (v, u), respectively (u, v).

3 Algorithms

3.1 Local search without saving information

In this section we present an algorithm for the case when we are for looking for free registers at only one point in the program, denoted *Root*. Information is not saved between consecutive calls. As it will be explained below, the algorithm has the "gracious time-out" property.

```
algorithm Local Search
LS := Procedure (v, R)
      (* R is a set of registers for which
         there are free paths from Root
         to v. The procedure returns a
         subset of R determined to be
         not-free.*)
      Local NewlyDead, R', Rnew.
      Rnew := R − Visited(v);
      Visited(v) := Visited(v) ∪ R;
      if v is a leaf or Rnew = ∅ then
            NewlyDead := Rnew;
            return NewlyDead
      fi;
      NewlyDead := ∅;
      for x ∈ Succ(v) do
            R' := Rnew;
            for i ∈ Rnew do
                  if read(i, (v, x)) then
                        NewlyDead :=
                              NewlyDead ∪ {i};
                        R' := R' \ {i}
                  fi;
                  if write(i, (v, x)) then
                        R' := R' \ {i} fi;
            od;
            NewlyDead :=
                  NewlyDead ∪ LS(x, R');
            Rnew := Rnew \ NewlyDead;
      od;
      return NewlyDead
end
```

The calling sequence for this procedure is

```
for v ∈ V do Visited(v) := ∅ od;
Free :=
      AllRegisters \ LS(Root, AllRegisters);
```

To prove the correctness of the algorithm above we start by proving two lemmas.

Lemma 1 *$LS(Root, AllRegisters)$ equals the set of all the registers ever included in NewlyDead within all the recursive calls.*
Proof: Obvious. □

Lemma 2 *For all nodes x, and for every register $i \notin LS(Root, AllRegisters)$ such that there exist a clean path with respect to i from Root to x, at termination $i \in Visited(x)$.*

Proof: By induction on the length of the paths. Fix a particular register $i \notin LS(Root, AllRegisters)$ and assume a clean path with respect to i from *Root* to x. The condition holds at *Root*. Assume that it holds for x's predecessor on the path, v. Then there was an invocation $LS(v, R_v)$ with $i \in R_v$. Consider the first such invocation. In this invocation $i \in R_{new}$. By the previous Lemma, i is never added to *NewlyDead*, and thus it is not removed at line 22 from R_{new}. Since (v, x) is clean, i is not removed from R' in either line 17 or line 19. Thus eventually there was an invocation $LS(x, R')$ in line 21 with $i \in R'$. □

Theorem 1 *Register i belongs to $LS(Root, AllRegisters)$ at termination if and only if i is in use at Root.*

Proof: For the "if" part assume that there is a clean path with respect to i from *Root* to some node v, and either an edge (v, x) with $read(i, (v, x))$ or v is a leaf. We must prove that $i \in LS(Root, AllRegisters)$. Assume not. Then by Lemma 2, at termination, $i \in Visited(v)$; hence there was some invocation $LS(v, R_v)$ with $i \in R_v$. If v is a leaf then i was added to *NewlyDead* in line 8; otherwise it was added to *NewlyDead* in line 16. But then by Lemma 1, at termination $i \in LS(Root, AllRegisters)$. Contradiction.

For the "only if" part, we claim that for each invocation $LS(v, R_v)$ the following invariant holds at line 3:

> R is a set of registers for which there are clean paths from *Root* to v.

We prove this by induction on the depth of the recursion. It clearly holds for $v = Root$. Consider an invocation $LS(x, R_x)$ with $x \neq Root$. By the induction hypothesis, the invariant holds for the invocation $LS(v, R_v)$ where v is the predecessor of x in the recursion stack. Thus we know that there is a clean path from *Root* to v with respect to each register $i \in R_v$. The tests in lines 14 to 20 imply that if $r \in R'$ at line 21 then the edge from v to x is clean with respect to r. But $R_x = R'$ and $R' \subset R_v$ and therefore there exists clean paths from *Root* to x with respect to each of the registers in R_x.

Returning to the "only if" part, note that if at termination register $i \in LS(Root, AllRegisters)$, then there was some invocation $LS(v, R_v)$ with $i \in R_v$ such that i was added to *NewlyDead* at line 16 or line 8. By the invariant, there exists a clean path from *Root* to v followed by a $Read(i)$ or v is a leaf. Hence the claim follows. □

Observations:

1. Our algorithm tends to find quickly registers whose freedom is easy to deduce. Since in many instances we need few free registers we can time-out the algorithms during execution. To this end, we add to the algorithm the line

4.5. **if TimeOut then return** R **fi**

With this modification $LS(Root, AllRegisters)$ becomes a superset of the registers that are in use at *Root*.

2. Since we are interested only in the registers that are free at *Root*, a register discovered to be "in-use" is removed from consideration as soon as possible (in line 22).

3. The obvious way to determine whether i is in use at *Root* is to do a DFS from *Root* using only edges that are clean with respect to i. If either a $Read(i)$ or a leaf is reached then i is in use at at *Root*. Note that if $i \in Visited(v)$ then v would be reached by the DFS algorithm as well. Thus our algorithm makes no more edge traversals than the total number of traversals made by the DFS algorithms and $Visited(v)$ represents a subset of the DFS trees that include v.

3.2 Local search with information saving

In some situations we are interested in free registers at more than one location. In this case we show how to save some of the information gathered in earlier runs, namely we will remember which registers were already determined to be in use at a particular vertex. These registers will be stored in an array denoted $Dead(v)$.

1. **algorithm** Local Search 2
2. $LS2$:= **Procedure** (v, R)
3. (* R is a set of registers for which there are free paths from *Root* to v. The procedure returns a subset of R determined to be not-free.*)
4. **Local** *NewlyDead*, R', R_{new}.
5. $R_{new} := R - Visited(v)$;
6. $Visited(v) := Visited(v) \cup R$;
7. **if** v is a leaf or $R_{new} = \emptyset$ **then do**
8. $Dead(v) := Dead(v) \cup R_{new}$;
9. **return** $R \cap Dead(v)$
10. **fi**;
11. **for** $x \in Succ(v)$ **do**
12. $R' := R_{new}$;
13. **for** $i \in R_{new}$ **do**
14. **if** $read(i, (v, x))$ **then**
15. $Dead(v) := Dead(v) \cup \{i\}$;
16. $R' := R' \setminus \{i\}$
17. **fi**;
18. **if** $write(i, (v, x))$ **then** $R' := R' \setminus \{i\}$ **fi**;
19. **od**;
20. $Dead(v) := Dead(v) \cup LS2(x, R')$;
21. **od**;
22. **return** $R \cap Dead(v)$
23. **end**

The calling sequence for this procedure is

```
for v ∈ V do Visited(v) := ∅;
    Dead(v) := ∅; od;
LS2(point_1, AllRegisters);
for v ∈ V do Visited(v) := Dead(v); od;
LS2(point_2, AllRegisters);
for v ∈ V do Visited(v) := Dead(v); od;
LS2(point_3, AllRegisters);
for v ∈ V do Visited(v) := Dead(v); od;
...
```

To prove correctness, we start from the following

Observation 1 *For any vertex x*

$$LS2(x, R) \subseteq R \cap Dead(v).$$

Lemma 3 *At all times, for all nodes v,*

(a) $Dead(v) \subseteq Visited(v)$

(b) If $i \in Dead(v)$ then i is in use at v.

Proof: For (a) observe that whenever a register i is added to $Dead(v)$ it already belongs to $Visited(v)$.

We prove (b) by contradiction. Consider the first time the claim is violated with respect to a certain register i and a certain node v. There are three cases to consider based on where i is added to $Dead(v)$.

1. Line 8: Then v must be a leaf.
2. Line 15: Then the path (v, x) fulfills (b).
3. Line 20: Since no previous violations occurred, if $i \in LS2(x, R')$ then by the observation above, $i \in Dead(x)$, hence i is in use at x. We claim that the edge (v, x) is clean with respect to i since otherwise i would have been removed from R' in line 16 or 18 contradicting $i \in LS2(x, R') \subseteq R'$. Thus i is in use at v.

□

Lemma 4 *$LS2(Root, AllRegisters)$ equals the set of all the registers returned by any induced recursive calls to LS2.*

Proof: Note that if an invocation $LS2(v, R)$ recursively calls $LS2(x, R')$ then $LS2(v, R) \supseteq LS2(x, R')$ because $LS2(x, R') \subseteq R$ and is added to $Dead(v)$. □

Lemma 5 *Let Root be one of the points where LS2 is invoked. For all nodes v, for all $i \notin LS2(Root, AllRegisters)$ and such that there exist a clean path with respect to i from Root to v, at the termination of the procedure $LS2(Root, AllRegisters)$ the register i is contained in the set $Visited(v)$.*

Proof: By induction on the length of the paths. Fix a particular register $i \notin LS2(Root, AllRegisters)$ and assume a clean path with respect to i from $Root$ to v. The condition holds at $Root$. Assume that it holds for v's predecessor on the path, w. Then there was an invocation $LS2(w, R_w)$ with $i \in R_w$. Consider the first such invocation. If at this stage $i \in Dead(v)$ then $i \in LS2(w, R_w) \subseteq LS2(Root, AllRegisters)$, contradiction. Otherwise, $i \notin Dead(v)$ and since w is not a leaf and $i \in R_{new}$, we execute line 11, with $v = w$, and $x = v$. Furthermore, since the relevant edge is clear, the tests in lines 13–19 do not remove i from R'. Thus the recursive invocation at line 20 leads to i being added to $Visited(v)$. □

Theorem 2 *Let Root be one of the points where LS2 is invoked. Then register i belongs to $LS2(Root, AllRegisters)$ at termination if and only if i is in use at at Root.*

Proof: For the "if" part assume that there is a clean path with respect to i from $Root$ to some node v, and either v is leaf or there is an edge (v, x) with $read(i, (v, x))$. We must prove that $i \in LS2(Root, AllRegisters)$. Assume not. If prior to the call to $LS2(Root, AllRegisters)$ we have $i \in Dead(y)$ for all nodes y on the clean path then $i \in Dead(Root)$ and clearly $i \in LS2(Root, AllRegisters)$. Otherwise fix y to be the node farthest from $Root$ on the clean path such that $y \notin Dead(y)$ prior to the call. By Lemma 5, at termination, $i \in Visited(y)$, so there was a first invocation $LS2(y, R_y)$ with $i \in R_y$. Within this invocation $i \in R_{new}$. If $y = v$ then i is added to $Dead(y)$ in line 8 or 15. Otherwise the recursive call to the successor of y on the path adds i to $Dead(y)$. Thus $i \in LS2(y, R_y) \subseteq LS2(Root, AllRegisters)$. Contradiction.

For the "only if" part, we claim that for each invocation $LS2(v, R_v)$ the following invariant holds at line 3:

> R is a set of registers for which there are clean paths from $Root$ to v.

We prove this by induction on the depth of the recursion. It clearly holds for $v = Root$. Consider an invocation $LS2(x, R_x)$ with $x \neq Root$. By the induction hypothesis, the invariant holds for the invocation $LS2(v, R_v)$ where v is the predecessor of x in the recursion stack. Thus we know that there is a clean path from $Root$ to v with respect to each register $i \in R_v$. The tests in lines 13 to 19 imply that if register $j \in R'$ at line 20 then the edge from v to x is clean with respect to j. But $R_x = R' \subseteq R_v$ and therefore there exists clean paths from $Root$ to x with respect to each of the registers in R_x.

Returning to the "only if" part, note that if at termination register $i \in LS2(Root, AllRegisters)$, then
(a) $i \in Dead(Root)$ before the call.
(b) There in an edge $(Root, x)$ with $read(i, (Root, x))$.
(c) There was an invocation $LS2(v, R_v)$ returning i with $v \neq Root$.

In case (a) the claim follows from Lemma 3(b). In case (b) the edge $(Root, x)$ implies the claim. In case (c) we have $i \in R_v \cap Dead(v)$ and by the invariant and Lemma 3(b), there exists a clean path from $Root$ to v and i is in use at v, hence i in use at $Root$. □

3.3 Global Search

We can use similar techniques to find the free registers at all points within a program via a "backwards" exploration of the entire graph. In this section we assume an imaginary $EndRoot$ such that every point in the program can reach the $EndRoot$. If this is not the case (e.g. infinite loops or leaves) we add imaginary edges to fulfill this condition. The added edges from points in infinite loops are clean with respect to all registers, while the edges added from leaves satisfy $Read(i)$ with respect to every register i.

1. **algorithm** Global Search
2. $GS :=$ **Procedure** (v, R)
3. (* R is a subset of the registers in use at v. *)
4. **if** $R \subseteq \mathit{Visited}(v)$ **then return fi**;
5. $\mathit{Visited}(v) := \mathit{Visited}(v) \cup R$
6. **for** $x \in \mathit{Pred}(v)$ **do**
7. $R' := R$;
8. **for** $i \in \mathit{AllRegisters}$ **do**
9. **if** $\mathit{write}(i, (x, v))$ **then** $R' := R' \setminus \{i\}$;
10. **if** $\mathit{read}(i, (x, v))$ **then** $R' := R' \cup \{i\}$;
11. **od**
12. $GS(x, R')$
13. **od**
14. **end**

The calling sequence for this procedure is

1. **for** $v \in V$ **do** $\mathit{Visited}(v) := \emptyset$; $Dead(v) := \emptyset$; **od**;
2. $GS(\mathit{EndRoot}, \emptyset)$

Theorem 3 *At termination register i belongs to $\mathit{Visited}(v)$ if and only if i is in use at v.*

Proof: We claim that the algorithm maintains the the following invariant at line 3:

R is a subset of the registers in use at v.

We prove the invariant by induction on the depth of the recursion. Initially, $R = \emptyset$ and the invariant holds. Assume that it holds for an invocation $GS(v, R)$, that calls $GS(x, R')$. Consider some $i \in R'$; there are two cases to consider:

- $i \in R$ and $\neg \mathit{write}(i, (x, v))$ – then inductively i is in use at v and therefore at x.
- $i \notin R$ and $\mathit{read}(i, (x, v))$ – then obviously i is in use at x.

Thus the invariant holds for $GS(x, R')$.

Returning to the main proof, the invariant implies the "only if" part since $\mathit{Visited}(v)$ consists only of registers that were in R for some invocation $GS(v, R)$.

For the "if" part, consider a register i in use at x; we use an induction on the length of the clean path from x to the edge e with $\mathit{read}(i, e)$. If this paths consists of only one edge (x, v) with $\mathit{read}(i, (x, v))$ then there is a first invocation involving v, say $GS(v, R)$, where in line 10 i will be added to R' and then $GS(x, R')$ will be called which will add i to $\mathit{Visited}(x)$, if not already there. If the path has length greater than 1, then let (x, v) be the first edge on the path. Inductively $i \in \mathit{Visited}(v)$. Hence there was a first invocation $GS(v, R)$, with $i \in R$ and since $\neg \mathit{write}(i, (x, v))$, register i is not removed from R' in line 9, and therefore there will be a call $GS(x, R')$ with $i \in R'$ which will add i to $\mathit{Visited}(x)$, if not already there. $\square$

Acknowledgment

We wish to thank Mike Burrows for introducing us to this problem and for many stimulating discussions and Greg Nelson and Lyle Ramshaw for their helpful comments that improved our exposition.

References

[1] A. V. Aho, J. E. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] R. Johnson and K Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78-79, 1993. Published as ACM SIGPLAN Notices 28(6).

Biography

Monika Rauch Henzinger received her Ph.D. from Princeton University in 1993. Afterwards, she was an assistant professor in Computer Science at Cornell University. She joined the Digital Systems Research Center in 1996.

Her main research interests are efficient algorithms and data structures, performance monitoring, and code optimization.