A Reference Chain Approach for Live Variables

Michael P. Gerlek[*], Michael Wolfe, Eric Stoltz
Oregon Graduate Institute of Science & Technology
Department of Computer Science and Engineering
P.O. Box 91000
Portland, OR 97291-1000 USA
503-690-1121 x7404
fax: 503-690-1548
`gerlek@cse.ogi.edu`

April 16, 1994

Abstract

The classical dataflow approach to determine the set of variables that are live at some point in a program entails using an iterative algorithm across the entire program, usually with costly bit-vector data structures. Methods based on sparse evaluation graphs exist, but these are solved on a per-variable basis and are also iterative. Inspired by our group's interest in sparse representations of use-def and def-use reference chains, we are investigating still another approach.

In this paper, we present an algorithm for determining liveness of variables based on upwards exposed use information at control flow branch points, analogous to the collection of reaching definition information in the popular Static Single Assignment form. We demonstrate the use of our technique with two applications, building interference graphs and eliminating dead code, and show that this new reference chain approach has important advantages over previous methods.

Keywords: *dataflow analysis*, *static single assignment*, *sparse evaluation graph*, *slotwise analysis*, *live variables*, *interference graph*, *dead code elimination*

[*]This research was supported by ARPA under grant F3062-92-C-135.

1 Introduction

A variable v at some point p in a program is considered *live at p* if v may be used along some control flow path from p, that is if v has an *upwards exposed use* [1]. If v is used nowhere after p, then v is *dead at p*. This determination of the set of live variables at any point in a program, called *live variables analysis*, is used by compilers in order to perform such analyses as the construction of *interference graphs* for register assignment (if two variables are both live at p they cannot share a register and are said to *interfere*) and such optimizations as *dead code elimination* (if v is defined at p but not subsequently used, i.e. v is dead after p, the defining statement may be removed).

Live variables analysis is a textbook backwards dataflow problem, and the standard solution requires an iterative algorithm. For each basic block b in a program, two sets are computed: the set of variables defined prior to being used in b, def_b, and the set of variables used prior to any definition, use_b. To compute which variables are live at the entry and exit to b, the following equations are used to combine the information at b with the information of the blocks following b:

$$\begin{aligned} in_b &= use_b \cup (out_b - def_b) \\ out_b &= \bigcup_{s \in succ(b)} in_s \end{aligned}$$

The computation of the *in* and *out* sets is iterated until convergence.[1]

Although this solution usually works well in practice, there are two drawbacks. First, the algorithm is iterative in nature. While the number of iterations for liveness analysis is small in most cases, other similar dataflow problems may iterate more times or may require more expensive equations to be solved at each block. In most cases, iteration is needed only for certain regions of the program. We are interested in solutions that will correspondingly minimize (or eliminate completely) the costs of this iterative approach.

The second drawback is the amount of storage required by the algorithm. Four sets must be stored for each block in the program, resulting in $\frac{VS}{8}$ 32-bit words of space if using bit-vectors, where V is the number of blocks and S the number of symbols in the program. If the bit-vectors are sparse, this is not an efficient technique; Choi et al note that "compiler writers generally acknowledge bit-vectors are overly consumptive of space" [3]. A preferable approach would be to take advantage of the definition/use information already within the program, avoiding the space required for the *def* and *use* sets.

Our research group's interest in the advantages of *reference chains* (representations of factored use-def and def-use chains [4]) has led to the investigation of an alternative to the iterative approach. This paper presents a solution to the live variables problem which uses λ-operators to merge upwards exposed reference information at branch points in the control flow graph. The idea is derived from the traditional Static Single Assignment (SSA) form in which ϕ-operators are placed at flow graph confluence points to merge multiple reaching definitions [5].

By using the information embedded within the flow graph at the λ-operators, we avoid the need for the *in* and *out* sets: at each block, the relevant information is determined by the confluence of the upwards exposed references determined by the λ-operators. As we will show, liveness analysis does not require explicit computation of these sets for such applications as interference graph construction and dead code elimination (and in fact improves the elimination algorithm). While our technique is admittedly slower and requires an equivalent amount of space as compared to the iterative bit-vector approach, it represents a first sparse graph solution to the live variables problem.

In the remainder of this section, we present some perspective and background material, relating our work to sparse evaluation graphs and the classical SSA form. In the next two sections, we present an

[1]We will assume the reader is familiar with these concepts; details may be found in most compiler texts [1, 2].

algorithm to construct the reference chains and an algorithm to compute liveness using them. In section 4 we show how to construct an interference graph from these chains and discuss other applications such as dead code elimination. In section 5 we present a performance analysis based on our implementations of these algorithms. We conclude in section 6 with some remarks on the implications of this work.

1.1 Sparse Graph Representations

Several alternative methods of dataflow analysis have recently been suggested, attempting to address the drawbacks of the traditional approach.

Sparse Evaluation Graphs. In practice, many nodes in a flow graph may not contribute to the final solution. In live variables analysis, for example, a given variable will be used or defined only in a small number of places in the program. *Sparse evaluation graphs* represent a subset of the flow graph for a particular problem, connecting only the nodes effecting the solution [3]. A standard solver is then applied to the reduced graph and the solution is mapped back to the original flow graph. Sparse evaluation graphs do not present a new solution technique as such but rather minimize the cost of the standard ones. The problem with these sparse graphs is that the dataflow problem must be solved *per variable*, requiring construction of a separate sparse graph for each variable. An open question is whether the cost of constructing n sparse graphs and solving n small problems is less than the cost of solving n (possibly vectorizable) problems simultaneously.

Static Single Assignment Form. SSA form is used to capture reaching definition information at points in the program [5]. The program is augmented with ϕ-operators which merge reaching definitions at flow graph confluence points. Uses are then linked to the unique reaching definition so that in this form the information (reaching definitions) is available only where required. SSA-based algorithms use these links in the program essentially like sparse evaluation graphs. SSA form only provides def-use links and cannot be used to solve all dataflow problems, as opposed to the general sparse evaluation graph mechanism: for example, while SSA form can be used to detect dead code, it cannot be used to construct interference graphs.

Slotwise Analysis. For dataflow problems requiring only a binary lattice ($\top$and $\bot$), there are only three interesting transfer functions that can occur at a node: either the node will have a known value of $\top$or $\bot$, or the node will propagate the solution of a predecessor node (for forwards problems) [6]. *Slotwise* analysis proceeds by first processing nodes with constant solutions. For nodes with the propagate transfer function, those that are reached by any nodes whose solution is $\bot$ will have $\bot$ as their solution. The remaining nodes will have $\top$ as their solution. The solution at a constant node does not depend on input from its predecessor, making the problem *strictly monotonic*, *i.e.* the solution at any point can only be lowered. The solution can be determined in one pass over the graph. Similar to sparse evaluation graphs, the slotwise approach is also performed "per variable".

The method we present here combines aspects of all of these approaches. Using λ-chains, the dataflow problem is solved on a sparse graph similar to SSA form, where liveness information is only represented at the points in the flow graph where it is needed. The dataflow problem is solved using a slotwise approach, where the λ-operators correspond to the nodes in the sparse graph. The solutions for all variables can be determined simultaneously, however, using an embedded graph rather than separate graphs.

2 Lambda Chain Construction

We begin by defining the framework for our representation and some necessary terms and concepts.

A program is represented as a *control flow graph* (CFG), $G_{CFG} = \langle V, E, \mathit{Entry}, \mathit{Exit} \rangle$, where V is a set of nodes representing basic blocks in the program and E is a set of edges representing sequential control flow. *Entry* and *Exit* are distinguished nodes representing the unique entry and exit points in the program; we assume all nodes are reachable from *Entry* and all nodes reach *Exit*. We also assume the placement of the so-called *technical edge* from *Entry* to *Exit*, needed for control dependence relations [7]. We denote an edge in the CFG from node X to Y as $X \to Y$. The set *Pred(X)* represents the predecessors of X in the CFG, that is the set of all nodes $v \in V$ such that $v \to X$. The set *Succ(X)* similarly represents the successors of X in the CFG.

Each block contains a list of operations or statements, denoted *Ops(v)*. Without loss of generality, we will assume each operation to be either a use (fetch), a definition (store), or a λ-operator for some symbol $s \in S$ where S is the set of all symbols in the program. We denote the type of some operation t as $\mathit{type(t)} \in \{\mathtt{use}, \mathtt{def}, \lambda\}$ and the symbol it references as $\mathit{symbol(t)} \in S$. If $\mathit{symbol}(t) = s$ for some t, then t *references* s.

Our technique uses the concepts of *post-dominator* and *post-dominance frontier*, which correspond to the concepts of *dominator* and *dominance frontier* but for the reverse of the control flow graph. Briefly, if for some nodes $X, Y \in V$, Y appears on every path from X to *Exit*, then Y post-dominates X, written Y *pdom* X. If Y *pdom* X and $Y \neq X$, then Y *strictly* post-dominates X, written Y *spdom* X. The *immediate* post-dominator of X, *ipdom(X)*, is the closest strict post-dominator of X. The *post-dominator tree* contains the set of nodes V, connected by edges such that $Y \to X$ in the tree iff $\mathit{ipdom}(Y) = X$.

The post-dominance frontier of a node Y, $\mathit{PDF}(Y) \subset V$, is the set of nodes X such that Y post-dominates a successor of X but does not strictly post-dominate X:

$$\mathit{PDF}(Y) = \{X \mid (\exists S \in \mathit{Succ}(X))(Y \mathit{pdom} S) \wedge \neg(Y \mathit{spdom} X)\}$$

The *iterated post-dominance frontier* of Y, $\mathit{PDF}^+(Y)$, is the limit of the sequence

$$\begin{aligned} \mathit{PDF}^1(Y) &= \mathit{PDF}(Y) \\ \mathit{PDF}^i(Y) &= \mathit{PDF}(Y \cup \mathit{PDF}^{i-1}(Y)) \end{aligned}$$

The set $\mathit{PDF}^+(Y)$ is computed via a worklist algorithm, and is equivalent to the iterated join set of the reverse control flow graph; we will rely on this in the following algorithms, and the reader is encouraged to refer to the seminal papers by Cytron *et al* for details [7, 5].

As an example, consider the following program:

```
i = 0
while (p) do
   if (q) then
      i = i + 1
   endif
endwhile
```

For clarity, we will make the basic blocks explicit and consider only references to `i`, distinguishing them by subscripting:

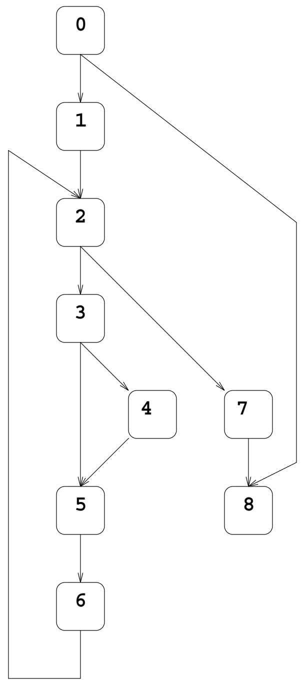

Figure 1: CFG for example program

```
L0:
L1:  i0 =
L2:  if (...) goto L7
L3:  if (...) then
L4:  = i1
     i2 =
L5:  endif
L6:  goto L2
L7:  ...
L8:
```

The corresponding control flow graph is shown in Figure 1 (note the placement of the technical edge $0 \rightarrow 8$) and the immediate post-dominators and post-dominance frontier sets are given in Table 1. For reference, the *def*, *use*, *in*, and *out* sets are also shown.

2.1 Chaining Algorithm

There are two phases involved in converting a program to the λ-chain form: the placement phase, which places the λ-operators at branch points in the CFG, where upwards exposed reference information must be combined, and the chaining phase, which creates the links (chains) from each λ-operator to its upwards exposed references. The algorithms for these two phases are directly adapted from the SSA

v	*ipdom* of v	$PDF(v)$	def_v	use_v	in_v	out_v
0	8					
1	2	0	i			
2	7	0,2			i	i
3	5	2			i	i
4	5	3		i	i	i
5	6	2			i	i
6	2	2			i	i
7	8	0				
8						

Table 1: Post-dominator and liveness for example program

construction algorithms of Cytron *et al* [7].

In the placement phase, each symbol in the program is processed separately as in SSA. First, a worklist is initialized to the set of blocks containing at least one reference to the symbol. A block w is selected from the worklist, and a λ-operator is appended to the list of operations of each block in the post-dominance frontier of w. Each block in the frontier is added to the worklist, creating the iterated post-dominance frontier set. This process iterates until the worklist is empty and all necessary λ-operators have been placed.

In the chaining phase, the arguments to λ-operators must be set such that each λ-operator is "linked" to the upwards exposed references of its symbol; the λ-operators, placed at CFG merge points, represent the combining of this information exposed along the outgoing paths. For each successor of the block a λ-operator is placed in, the λ-operator has exactly one link, initially empty ($\diamond$). These links are now filled in to point to the upwards exposed reference of the λ-operator's symbol. The blocks in the CFG are visited in a bottom-up order (we use the post-dominator tree), saving the most current reference to each symbol at each referencing operation. When the top of the block is reached, links in the λ-operators of predecessors of the current block are set to point to the current, *i.e.* upwards exposed, reference.

Consider the example from above. The worklist is initialized to blocks 1 and 4; since $PDF^+(1) \cup PDF^+(4) = \{0, 2, 3\}$, a λ-operator is placed at the end of each of these three blocks.

```
L0:  i_3 = λ(◇,◇)
L1:  i_0 =
L2:  if (...) goto L7
     i_4 = λ(◇,◇)
L3:  if (...) then
     i_5 = λ(◇,◇)
L4:  = i_1
     i_2 =
L5:  endif
L6:  goto L2
L7:  ...
L8:
```

In the second phase, the current reference is initially $\emptyset$. The CFG is traversed in the order {8, 7, 2, 1, 6, 5, 4, 3, 0}. Starting with block 8, the λ-operator in predecessor block 0 has its second link set to

$\diamond$. At block 7, predecessor 2 has its second link set to $\diamond$. Visiting block 2 next, the current reference is set to $\mathtt{i}_4$. At block 1, the current reference is set to $\mathtt{i}_0$ and the old reference to $\mathtt{i}_4$ is saved. The first link of the λ-operator in predecessor block 0 is set to $\mathtt{i}_0$. We must return down the post-dominator tree: in block 1 the current reference is restored to $\mathtt{i}_4$. The process continues at block 2 by visiting its other child, block 6. After all blocks have been visited and all links have been processed, the resulting program is:

```
L0:  i3 = λ(i0, ◇)
L1:  i0 =
L2:  if (...) goto L7
     i4 = λ(i5, ◇)
L3:  if (...) then
     i5 = λ(i4, i1)
L4:     = i1
        i2 =
L5:  endif
L6:  goto L2
L7:  ...
L8:
```

Algorithm 1 *Lambda Chain Construction*

Given:	V, the set of basic blocks in the program,
	Exit, the exit block, and
	S, the set of symbols in the program,
Do:	insert λ-operators and create the corresponding chains

We will use several data structures. For each block $v \in V$, two fields are used: *v.InWork*, the symbol for which block v was last inserted into the worklist, and *v.Added*, the symbol for which block v last had a λ-operator added. For each operation $t \in Ops(v), \forall v \in V$, one field is used: *t.Save*, the operation that was the last reference for *symbol(t)*. For each symbol $s \in S$, two fields are used: *s.UpExpRef*, the set of blocks containing upwards exposed references to s, and *s.CurrRef*, the operation that is the current reference to s.

```
procedure Construct_Chains(V, Exit, S)
    Place_Lambdas(V, S)
    Set_Links(Exit)
endprocedure
```

The code for procedures **Place_Lambdas** and **Set_Links** are shown in Figures 2 and 3. Note that at line 15 of **Place_Lambdas**, the links for the new λ-operator are assumed to be empty. The post-dominance frontiers are assumed to have already been created. □

2.2 Correctness

The proof of correctness for λ-chain construction can be directly adapted from the original SSA construction proofs by Cytron *et al* [7]. In the SSA ϕ-placement algorithm, a ϕ-operator for symbol s is placed at the iterated dominance frontier[2] of the set of blocks containing assignments to s. In the

[2]Technically, the iterated join set is used; the iterated dominance frontier is proved equivalent.

```
procedure Place_Lambdas($V$, $S$)
    for $v \in V$ do
        $v.InWork = \emptyset$
        $v.Added = \emptyset$
    endfor
    $Worklist = \emptyset$

    for $s \in S$ do
        for $w \in s.UpExpRef$ do
            $Worklist = Worklist \cup \{w\}$
            $w.InWork = s$
        endfor
        while $Worklist \neq \emptyset$ do
            remove some $w$ from $Worklist$
            for $p \in PDF(w)$ do
                if $p.Added \neq s$ then
                    append a $\lambda$-operator for $s$ at end of $Ops(p)$
                    $p.Added = s$
                    if $p.InWork \neq s$ then
                        $WorkList = WorkList \cup \{p\}$
                        $p.InWork = s$
                    endif
                endif
            endfor
        endwhile
    endfor
endprocedure
```

Figure 2: Algorithm for inserting λ-operators

```
procedure Set_Links($v$)
    for $t \in Ops(v)$ in reverse order do
        $s = symbol(t)$
        $t.Save = s.CurrRef$
        $s.CurrRef = t$
    endfor

    for $p \in Pred(v)$ do
        for $t \in Ops(p)$ where $type(t) = \lambda$ do
            $s = symbol(t)$
            set corresponding $\lambda$-argument to $s.CurrRef$
        endfor
    endfor

    for $p \in PDomChild(v)$ do
        Set_Links($p$)
    endfor

    for $t \in Ops(v)$ in forward order do
        $s = symbol(t)$
        $s.CurrRef = t.Save$
    endfor
endprocedure
```

Figure 3: λ-operator Linking Algorithm

Place_Lambdas algorithm, we have replaced the notion of dominator with post-dominator and extended the set of blocks containing assignments to s with the set of blocks containing references to s.

Similarly, our procedure **Set_Links** performs the same task as the SSA *SEARCH* procedure. For each successor at a branch point in the CFG, a link of a λ-operator at that CFG node is set to the upwards exposed reference along that path, as the links of ϕ-operators are set to the corresponding reaching definitions along incoming branches.

By this reasoning, **Construct_Chains** inserts λ-operators at precisely those points where dataflow information must be represented. In the Static Single Assignment form, a ϕ-operator represents the set of the reaching definitions of some variable at that CFG confluence point. In our representation for liveness analysis, a λ-operator represents the set of upwards exposed references for some variable at that CFG branch point.

3 Liveness

Once the λ-operators have been placed and their links set, as described in Section 2, the *in* and *out* live sets may be determined for each block in the program. This is accomplished by traversing the flow graph and adding and deleting variables from the live set at each operation.

3.1 Liveness Algorithm

The CFG is traversed in bottom-up order, starting at the exit block. At the bottom of *Exit*, the set of live variables is $\emptyset$, corresponding to out_{Exit}. The operations in the block are visited in reverse lexical order, modifying the live set at each point as needed: at a use the variable is added to the live set, if not already live, and at a definition the variable is removed from the live set, *i.e.* it goes dead. At the top of the block, the live set is precisely the in_{Exit} set.

The live set is carried to the next block to be visited, and the process of visiting each operation is repeated. At each λ-operator site, however, it must be determined whether the operator represents a use or a definition of its variable; this is a function of the nature of the links of the λ-operator. Intuitively, a λ-operator may be viewed as a placeholder, representing a "forwarding" of liveness information from some other point in the flow graph.

A simple lattice framework is used, with each operation assigned a lattice value representing its effect on its symbol. In this lattice,

$$\begin{tikzcd}\top \arrow[d, dash] \\ \bot\end{tikzcd}$$

the value $\top$ represents "dead" and $\bot$ represents "live". All definitions and uses are assigned lattice values of $\top$ and $\bot$, respectively. The lattice value of each λ-operator is initially set to $\top$. When a λ-operator is encountered in the reverse-order visit of operations, its links are examined. Each link may be undefined ($\diamond$), implying no upwards exposed reference of the symbol along that path, or it may point to a use, a definition, or another λ-operator for the symbol. The lattice value of a λ-operator is set to be the meet ($\sqcap$) of the lattice values of its links (where an undefined link is assigned a lattice value of $\top$). This value determines the λ-operator's effect on the live set, that is whether it should be treated as a use ($\bot$) or a definition ($\top$).

If the links of a λ-operator all point to uses or definitions or are undefined, this is straight-forward. If one or more of the links is to another λ-operator, another technique is needed, since the lattice value of the second λ-operator must be determined before it can be used. In general, the second λ-operator may lead to still other λ-operators or may even have a link back to the first λ-operator, creating a cycle. Such is the case in the example from the previous section. The lattice value of the λ-operator defining $\mathtt{i}_4$ in block 2 is the meet of $\top$ and the lattice value of the λ-operator in block 3. In terms of liveness information, this corresponds to the fact that `i` is dead outside the loop and its liveness is not known inside the loop.

A demand-driven technique is used to assure that all λ-operators are assigned lattice values before some other λ-operator requires them. The lattice value of each link of a λ-operator encountered during the bottom-up traversal of the flow graph is evaluated using Tarjan's well-known algorithm for detecting strongly connected components (SCCs) in directed graphs [8]. This algorithm has the important property that each component in the graph is visited and processed only after each of its descendants have been visited and processed. Tarjan's algorithm is applied to an abstraction of the program, the *λ-graph*, $G_\lambda = \langle V, E\rangle$, where V is the set of operations (uses, definitions, and λ-operators) in the program and E is the set of links from each λ-operator to other operations.

When the lattice value of a λ-operator is required, Tarjan's algorithm is used to "search" the λ-graph, starting at that root, and assign lattice values to all λ-operators the root "depends" on for its solution. When a component is identified, the nodes within it (a set of operations) are assigned a lattice value based on the lattice value of all the successors of the SCC. Note that no iteration is required to determine the lattice value of the cycle; the lattice value of each node in the component is set to the meet of the set of successors of the component.[3]

In Figure 4 we show the λ-graph for the example program, with the lattice value to the right of the nodes. To determine the lattice value of the λ-operator at block 3 ($\mathtt{i}_5$), for example, the cycle assumes the meet of the undefined second link of the λ-operator of $\mathtt{i}_4$ ($\top$) and the first link of the λ-operator of $\mathtt{i}_5$ to the use of $\mathtt{i}_1$, $\perp$. The meet, $\perp$, signifies `i` is live in the cycle, and therefore live at the bottom of blocks 2 and 3.

Algorithm 2 *Liveness*

<u>Given:</u>	*Exit*, the exit block in V
<u>Do:</u>	determine liveness at each block

The algorithm is shown in Figure 5. The set *Live* is initialized to $\emptyset$ and it is invoked as **Liveness(***Exit*, *Live***)**. The bottom-up walk is performed as a recursive traversal of the post-dominator tree.
Note that the sets *in* and *out* for each block, required by the iterative dataflow algorithm, are not explicitly computed. When the loop at line `1` has processed all the λ-operators (and not yet processed any other operations), *Live* is precisely out_v. When line `21` is reached, *Live* is precisely in_v. □

3.2 Correctness

Correctness of the liveness algorithm can be shown by induction. The solution (the *out* set) is trivially correct at the Exit block ($Live = \emptyset$). We will show (informally) that at each block subsequently visited, the correct *out* set is computed.

At the end of some block v, the set *Live* is determined by the state of the *Live* set at the top some block w that is a descendant in the flow graph, corresponding to the in_w set. The operations in v serve to add and remove symbols from the live set.

[3] This is an example of a *cluster partitionable* [9] or *uniformly monotonic* [10] problem. We omit details due to space limitations.

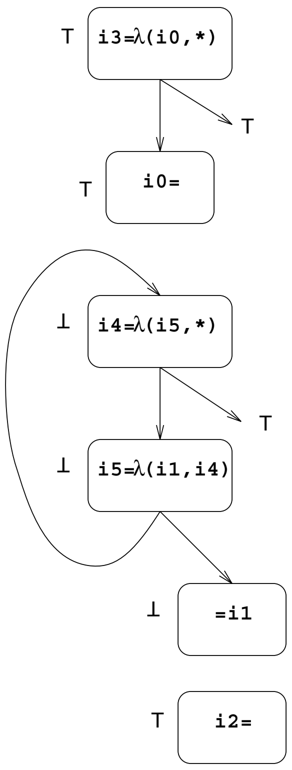

Figure 4: λ-graph for example program

```
procedure Liveness($v$, $Live$)
    for $t \in Ops(v)$ in reverse order do
        if $type(t) = \lambda$ then
            if $t.Visited$ = false then
                set $t.Lattice$ using Tarjan's alg on the $\lambda$-graph
                $t.Visited$ = true
            endif
        endif
        $t.Marked$ = false
        $s = symbol(t)$
        case $t.Lattice$
            $\perp$: if $s \notin Live$ then
                        $Live = Live \cup \{s\}$
                        $t.Marked$ = true
                  endif
            $\top$: if $s \in Live$ then
                        $Live = Live - \{s\}$
                        $t.Marked$ = true
                  endif
        endcase
    endfor

    for $p \in PDomChild(v)$ do
        Liveness($p$, $Live$)
    endfor

    for $t \in Ops(v)$ in forward order do
        $s = symbol(t)$
        case $t.lattice$
            $\perp$: if $t.Marked$ then $Live = Live - \{s\}$
            $\top$: if $t.Marked$ then $Live = Live \cup \{s\}$
        endcase
    endfor
endprocedure
```

Figure 5: Liveness Algorithm

If v is not immediately post-dominated by w, however, other blocks may contribute to the live set of v. That is, if v has more than one successor, the set *Live* must be modified to account for the presence of λ-operators. For each successor of v, each λ-operator in v will have one link to the subsequent reference (if any) of its variable along the path containing that successor. If the subsequent reference is a use, a definition, or no reference, the lattice value is set accordingly. If the subsequent reference is another λ-operator, the λ-graph will form a tree whose leaves are a use or a definition or are undefined (recall that by the algorithm used, cycles are effectively collapsed to a single λ-operator). Thus, irrespective of the flow graph, the lattice values inherited by the λ-operator will reflect the upwards exposed references of the symbol; there can be no other intervening references to the symbol in the program, by the construction in section 2.

The *Live* set is now modified by each use, definition, and λ-operator in the block. At the top of the block, the *Live* set is carried to the next block in the post-dominator tree. When returning to block v, only those operations that affected the *Live* set are undone, so that at the bottom of v the *Live* set is precisely equal to what it started as.

4 Applications of Liveness

4.1 Interference Graph Construction

Most modern compilers perform register assignment with graph coloring algorithms [1, 11]. An *interference graph* is constructed, such that $G_{IG} = \langle S, E \rangle$, where S is the set of symbols in the program and E is a set of edges such that $(r, s) \in E$ iff r *interferes* with s, that is, if r and s are simultaneously live at any point in the program. The nodes in the graph are then "colored", where each color assignment represents a mapping from a symbol (node) to a physical register.

Traditionally, live variable analysis is performed to compute the set of variables live *out* at each block. Then, as a separate pass, interferences are determined for each block. This is accomplished by first initializing a set *Live* with the set *out* and then visiting each operation in the block in reverse order. At each use of some symbol s in the block, s is added to *Live*. At each definition of s, s is marked as interfering with all variables currently in *Live* and s is removed from *Live*.

The algorithm presented in the previous section to compute liveness is easily modified to construct interference graphs in the same pass.

Algorithm 3 *Interference Graph Construction*

Given:	*Exit*, the exit block in V
	I, initialized to $I_{st} = 0, \forall s, t \in S$
Do:	determine the liveness at each block and compute the interferences

The algorithm is the same as Figure 5, with the addition of a line after line `11`:

```
11.1  if $t.Type \neq \lambda$ then $I_{sl} = 1, \forall l \in Live$
```

The interference graph is represented as a (symmetric) matrix I, such that $I_{rs} = 1$ if symbol r interferes with symbol s and $I_{rs} = 0$ otherwise. □

4.2 Dead Code Elimination

An assignment to a variable that is not subsequently used is termed *dead code* [1]. Dead (or *useless*) code usually occurs as a result of compiler optimization: a branch condition may be determined to be constant, code motion might effectively copy but not move computations, etc. Dead code elimination is an important optimization that is always beneficial and may be required several times during compilation.

The usual method for performing elimination is to visit the operations in each block, updating the *live* set as for traditional interference graph construction. When a definition of some symbol s is reached such that s is not in the live set, the definition and its right-hand side are removed. This will not account for the impact of the liveness of the variables on the removed right-hand side, however. Consider this case:

```
L1:  y = 1
     if (cond) then
L2:     x = y + 2
L3:  endif
```

where $out_1 = \{\texttt{y}\}$ and $out_2 = out_3 = \emptyset$. The assignment to `x` can be eliminated because it is not live after block 2, but the assignment to `y` cannot be eliminated: the set out_1 does not reflect the elimination. Good dead code detection requires dynamic live information of the sort provided by our approach to liveness.

Algorithm 4 *Dead Code Elimination*

Given:	*Exit*, the exit block in V
Do:	determine the liveness at each block and eliminate dead code

The algorithm is the same as Figure 5, with line `18` changed and two lines added:

```
  18  else
18.1        if $t.Type \neq \lambda$ then remove this def and its associated rhs
18.2  endif
```

□

4.3 Other Uses of Liveness Information

Liveness information may be used by a compiler for a few other relatively minor purposes; as these other problems are also amenable to our technique, they are included here for the sake of completeness.

Uninitialized variables: In most languages, the use of a variable prior to its definition is either an incorrect program or will result in undefined behavior. Such uses are easily detected if a (local) variable is live at *Entry*.

"Intent" determination: It may be useful to determine the type of usage of each formal parameter in a procedure – is it defined but not used, used but not defined, or both used and defined?[4] This information might be used when performing, for example, interprocedural constant propagation: if a variable known to be constant is passed as a reference parameter to some procedure that uses but does not define it, the variable will not be killed at the call site.

In certain cases the needed information may be determined trivially by the parser; in general, however, after any transformations have been applied by an optimizing compiler, full dataflow analysis is needed.

[4]This type of information may be reflected in some languages, such as the INTENT attribute in Fortran 90 [12].

5 Experimental Results

Three areas are of interest: the effects of the traditional, bit-vector iterative approach to liveness, the effects of our λ-chain method, and a time/space comparison of the two. In this section, we present experimental data in these areas. We will show that while the λ-chain algorithm for live variable analysis is no faster than the traditional method, it is still a competitive alternative.

The algorithms for constructing λ-chains, determining liveness, and constructing the interference graph have been implemented in Nascent, a prototype high-performance, restructuring Fortran 90/HPF compiler. The programs used to collect the data were taken from two well-known benchmark suites, the Perfect Club [13] and the Rice Compiler Evaluation Suite (RiCEPS). Each of these suites is made up of several Fortran 77 programs, representative of actual workloads found in high performance, scientific computation environments. Over 100,000 lines of source code are included.

Table 2 characterizes these suites. For each program, as well as for the entire suites, the total number of lines, program units (subroutines), basic blocks, and (referenced) symbols is given. The actual number of uses and definitions of all the symbols is also given. The data presented in this section was collected after an intermediate "lowering" phase within Nascent, so compiler temporary variables are included in the symbol counts. Variables used as subroutine arguments may be counted as uses and definitions at the call site, depending on the nature of the argument. Input/output statements were ignored, and no other optimizations were performed.

5.1 Data for the Traditional Approach

The iterative, bit-vector method for computing liveness information requires four sets be computed and and stored at each block in the program: the *in*, *out*, *use*, and *def* sets. In Table 3 the number of bits required (in thousands) *per set* is shown for each program. For the entire Perfect suite, this works out to about 300K 32-bit words of storage required for all four sets. The next four columns show that, of these four sets, the *in* and *out* sets are sparse. The table shows the percentage of the vectors actually used, about one percent for the entire suites. Approximately 40% of the bits in the *in* and *out* sets are used, however. Since no optimizations were performed, these densities are only an approximation. Application of transformations such as conversion to Static Single Assignment form or optimizations such as common subexpression elimination may decrease the number of live symbols relative to the total number of symbols.

The time required for convergence of the the solution is also shown in Table 3. Less than four iterations are required for computing the sets, on average, and never more than eight.

5.2 Data for the λ-chain Approach

The space required for the λ-chain method, relative to the number of symbols in the program, is shown in Table 4. Note that if a symbol is referenced (used or defined) only at the *Entry* block, it will require no λ-operators. For this reason, the number of referenced symbols requiring λ-operators in the entire Perfect and RiCEPS suites is 25351 and 21130, respectively, slightly lower than the total number of referenced symbols, 26027 and 21624. The table shows the number of references per symbol using the larger number and λ-operators created per symbol using the smaller. On average, the ratios are close: slightly more than five references per symbol, and the space required by the λ-operators tracks the number of references to the symbol. A histogram of the number of references (or λ-operators) per symbol yields a curve with a long tail, but, as indicated, 95% of the cases were at most only few times larger than the average.

program	*lines*	*units*	*blocks*	*syms*	*uses*	*defs*
ADM	6105	97	2694	3945	13701	4290
ARC2D	3964	39	1638	2316	11615	2790
BDNA	3977	43	2093	2386	9335	3395
DYFESM	7608	55	1189	999	2419	1006
FLO52Q	1986	28	1468	1863	7482	2791
MDG	1238	16	458	804	2410	910
MG3D	2812	28	1350	1422	9203	2488
OCEAN	4343	36	1277	1224	3832	1909
QCD	2327	35	1294	759	3222	1189
SPEC77	3885	65	2890	2224	9109	3093
SPICE	18521	128	7428	6803	30131	11579
TRACK	3784	32	970	887	3224	979
TRFD	485	7	537	395	1051	546
Perfect	61035	609	25286	26027	106734	36965
BOAST	8067	58	5810	5004	20980	6318
CCM	23556	145	5432	6405	24067	8202
HYDRO	13049	36	1250	1861	5758	2236
LINPACK	797	11	334	283	1112	484
QCD	2353	34	1274	926	3588	1470
SIMPLE	1313	8	521	581	2935	792
SPHOT	1144	7	431	390	1509	540
TRACK	3735	34	959	901	3254	969
WANAL1	2109	11	2161	1552	5173	2779
WAVE	7520	92	3371	3712	15828	4685
RiCEPS	63643	463	21543	21624	84204	28475

Table 2: Sample benchmark programs

	space	% of vector used				iterations	
program	10^3 *bits*	*use*	*def*	*in*	*out*	*avg*	*max*
ADM	174	4.0	2.2	40.7	40.7	3.5	7
ARC2D	135	3.2	1.8	27.5	27.9	3.9	5
BDNA	192	2.2	1.4	34.5	34.6	3.5	6
DYFESM	35	4.7	2.8	37.1	37.7	3.5	6
FLO52Q	142	2.3	1.6	37.7	37.9	3.8	5
MDG	35	3.3	2.1	51.0	51.0	3.8	6
MG3D	150	2.2	1.5	19.5	19.8	3.5	7
OCEAN	233	1.1	0.7	45.5	45.5	3.1	6
QCD	58	3.1	1.9	39.5	39.9	3.6	8
SPEC77	165	2.9	1.6	57.5	57.8	3.6	6
SPICE	1070	1.5	0.9	42.0	42.0	3.1	7
TRACK	44	3.2	1.9	45.1	45.1	3.4	6
TRFD	41	1.8	1.2	23.6	24.0	5.0	6
Perfect	2475	2.1	1.3	40.0	40.1	3.5	8
BOAST	932	1.4	0.7	34.8	34.9	4.4	6
CCM	824	1.5	0.9	33.5	33.6	3.0	6
HYDRO	150	2.0	1.3	37.7	37.8	3.2	5
LINPACK	15	4.0	2.6	32.9	32.9	3.2	4
QCD	76	2.5	1.7	41.6	41.7	3.5	8
SIMPLE	83	1.4	0.8	44.3	44.4	3.1	5
SPHOT	86	1.0	0.6	40.5	40.6	3.3	7
TRACK	45	3.2	1.9	45.1	45.0	3.3	6
WANAL1	2285	0.2	0.1	42.7	42.8	4.0	8
WAVE	342	2.1	1.2	29.3	29.3	3.0	5
RiCEPS	4839	0.9	0.5	38.5	38.6	3.3	8

Table 3: Traditional method data

	refs/sym		λ/sym	
Program	*avg*	95%	*avg*	95%
ADM	4.6	16	3.6	9
ARC2D	6.2	20	3.6	10
BDNA	5.3	18	3.6	10
DYFESM	3.4	12	3.3	12
FLO52Q	5.5	17	3.9	11
MDG	4.1	13	3.0	7
MG3D	8.2	26	5.9	20
OCEAN	4.7	15	3.6	10
QCD	5.8	21	4.8	13
SPEC77	5.5	17	4.0	10
SPICE	6.1	19	8.7	29
TRACK	4.7	14	6.0	27
TRFD	4.0	13	4.6	11
perfect	5.5	18	5.3	17
BOAST	5.5	16	6.8	21
CCM	5.0	15	4.4	12
HYDRO	4.3	11	4.2	10
LINPACK	5.6	20	3.2	9
QCD	5.5	19	4.4	12
SIMPLE	6.4	17	3.8	9
SPHOT	5.3	15	8.4	29
TRACK	4.6	14	5.9	27
WANAL1	5.1	8	8.7	12
WAVE	5.5	17	4.3	12
riceps	5.2	15	5.3	17

Table 4: λ-chain data

The total number of λ-operators created for the Perfect suite can be found by multiplying the number of λ-operators per symbol by the number of symbols, yielding approximately 134,000 λ-operators. The amount of space required to implement a λ-operator in a compiler's intermediate form is obviously implementation dependent, but the best case would be one word per operator, resulting in 130K words. This is of the same order of magnitude as in the traditional algorithm (300K).

To determine liveness, the method we have outlined requires processing strongly connected components in the λ-graph. We have found that on average slightly more than 40 percent of the λ-operators in the benchmarks were part of such a component; the percentages ranged from a low of 7 to a high of 75, and most were clustered between 30 and 50 percent. The component sizes ranged from 2 to 8, with an average of 4.2.

5.3 Comparative Performance

We next consider the speed of the iterative, bit-vector algorithm and the speed of different implementations of the λ-chain algorithm. The implementations were written with equivalent consideration for performance and extensive optimization was left to the compiler (gcc version 2.5.8, `-O2`). The experiments were run on three Sun SPARC computers: `A`, an 80MHz IPX workstation with 32MB; `B`, a 40MHz Classic workstation with 48MB; and `C`, a SPARCcenter 2000 server.

The implementation of the liveness algorithm presented in section 3 uses a recursive procedure to do the traversal of the post-dominator tree. The traditional liveness computation is easily implemented as an iterative procedure, however. Similarly, either a bit-vector or a linked list may be used in maintaining the *Live* set in the λ-chain algorithm. (The traditional algorithm uses bit-vectors for the four sets since the operations required by the dataflow equations "vectorize" as logical operations.) We found that, on the SPARC, the choice of iterative/recursive traversal and bit-vector/linked-list sets generally varied execution times for the interference graph construction by 10 to 20%, with the iterative linked-list approach performing the best.

The traditional algorithm for interference graph construction has three steps:

- *init*: compute the *use* and *def* sets
- *live*: compute the *in* and *out* sets
- *ifg*: compute the interference graph

The λ-chain algorithm also has three steps:

- *place*: insert λ-operators
- *chain*: set the links for the λ-operators
- *ifg*: compute liveness and the interference graph

In addition, both methods require some overhead in our implementation to allocate storage for the data structures used. The traditional algorithm performs less work than the λ-chain algorithm, and ultimately performs faster. In comparing the two initial phases of both methods, the traditional algorithm requires few iterations, performing simple operations at each block, whereas the λ-chain algorithm requires considerably more time to place the λ-operators for each symbol and then traverse the tree, maintaining the current reference to each symbol and setting links. In comparing the interference graph phase, the λ-chain algorithm uses a complex flow graph traversal method and requires two passes through each block (going up and going down). The traditional algorithm visits blocks in any order, relying on the static *out* sets, and requires only one pass. (Note the time to compute the post-dominance frontier is

	Perfect			*RiCEPS*		
	A	B	C	A	B	C
Traditional	72	137	68	123	225	115
init	3	5	2	3	5	2
liveness	1	2	<1	2	<1	<1
ifg	43	96	44	70	154	72
overhead	25	34	22	48	64	39
Lambda	119	190	123	197	306	206
placement	18	25	14	22	29	22
chaining	8	11	5	7	9	4
ifg	58	116	77	98	192	131
overhead	35	38	28	71	75	48
Ratio	1.65	1.39	1.81	1.60	1.36	1.79

Table 5: Times for traditional and λ-chain methods

not considered for the λ-chain algorithm because *PDF* sets are useful for other analyses (*e.g.* control dependence) and in any case will account for less than one percent of the total compilation time.)

Table 5 presents the time (in seconds) for each of these six phases, plus the overhead required. The total time for each method and the performance ratio are also shown. The data is presented for two runs, one for all of the programs in both suites (granularity precludes accurate per-program timing). The experiments were run on all three platforms, yielding different performance ratios as a result of the large amount of memory required by the algorithms and the very different memory configurations in each machine. The overall time required using λ-chains is roughly 60% greater than the traditional algorithm.

6 Conclusions

We have presented a new technique for determining liveness based on reference chains. The technique is of interest in four respects: (1) a sparse SSA-like representation is used which embeds the necessary definition/use information within the program at the only the points required; (2) liveness is not determined for each variable separately, but instead allows the solution of all variables in one pass; (3) the liveness of variables is dynamically computed, rather than storing sets each block in the flow graph, improving the precision of optimizations such as dead code elimination; and (4) liveness is determined without iteration, taking advantage of the partitionable nature of the dataflow problem.

The algorithms to compute the λ-chains and construct the interference graph were implemented in a compiler and shown to be competitive with the textbook dataflow approach in both space and time. While this new method does require more computation, its use by other analyses and optimizations may amortize the cost.

There are three areas that remain to be examined. Most importantly, the relationship between the ϕ-operators for reaching definitions in the Static Single Assignment form and our λ-operators for upwards exposed references suggests that this style of analysis – capturing dataflow information at join and branch control flow points – may provide a complete basis for a set of dataflow problems. The algorithms we have presented here provide evidence for this.

Second, in this paper we have considered the liveness property of variables, a backwards dataflow problem. A related problem is that of *busy* or *anticipatable* expressions: an expression e is said to be *busy* (*very busy*) at point p if e may (will) be used after p [1]. The use of operators similar to λ-operators, using expressions instead of variables, may be investigated.

Finally, the characteristics of the λ-operators inserted should be examined. In particular, λ-operators whose lattice value does not contribute to the solution need not be inserted. It remains to be seen if this can be determined during construction and how much benefit it would provide. A somewhat related question of great importance to optimizing compilers is whether λ-chains can be dynamically updated to reflect transformations within the program.

References

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

[2] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin-Cummings, Menlo Park, CA, 1988.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conf. Record 18th Annual ACM Symp. Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 1991.

[4] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proc. of 27th Annual Hawaii International Conference on System Sciences*, pages 43–52, January 1994.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing Static Single Assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, San Francisco, June 1992.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conf. Record 16th Annual ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989.

[8] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[9] Frank Kenneth Zadeck. Incremental data flow analysis in a structured program editor. In *Proc. SIGPLAN '84 Symp. on Compiler Construction*, pages 132–143, Montreal, Canada, June 1984.

[10] Michael Wolfe, Michael P. Gerlek, and Eric Stoltz. Demand-driven data flow analysis. (unpublished), 1994.

[11] Preston Briggs. Register allocation via graph coloring. PhD Dissertation COMP TR92-183, Rice Univ., Dept. Computer Science, April 1992.

[12] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook.* McGraw-Hill Book Company, New York, NY, 1992.

[13] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. In *International Conference on Supercomputing*, pages 254 – 266, March 1990.