Register Liveness Analysis of Executable Code *

Robert Muth
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
Phone: 520-621-2858
Fax: 520-621-4246
Email: muth@cs.arizona.edu

* This work was supported in part by the National Science Foundation under grant CCR-9711166

Abstract

Liveness analysis of variables is a well-understood technique employed by most compilers to guide optimizations such as useless code elimination and register allocation. Liveness analysis can also be performed on object code if we let registers take the place of variables. The increasing interest in systems that modify object code or executables has generated a need for a fast and accurate register liveness analysis. This paper shows how to accurately compute register liveness information in a time and space efficient manner and how to cope with irregular control flow not encountered in high level languages. Tradeoffs between the precision of the analysis and its computation time and space are discussed. In particular, context sensitive and context insensitive register liveness analysis are compared. Experimental results on precision, time and space usage are provided as well as the impact of liveness analysis on optimizations.

KEY WORDS: register liveness analysis; data flow analysis;

1 Introduction

Liveness analysis of variables is a well-understood technique employed by most compilers to guide optimizations such as useless code elimination and register allocation [9]. Liveness analysis can also be performed on object code if we let registers take the place of variables. The increasing interest in systems that modify object code or executables has generated a need for a fast and accurate register liveness analysis. On the one hand we have optimizing systems such as Alto [10], EEL [7], Etch [12], OM [14], Spike [1] which make use of the liveness information in a more classical way, ie. to reduce number of (executed) instructions. Instrumentation systems such as Atom [13], Pixie [15] and qp(t) [5] on the other hand, add new instructions to the executable in order to collect information about the executable at run time. Scratch registers needed by those new instructions are obtained by temporarily spilling registers to memory or by changing the compiler to not use certain registers in executables. The first approach slows down the instrumented executable significantly. In fact, the cost of spilling frequently exceeds the instrumentation cost itself [6]. The second approach yields suboptimal uninstrumented executables. As we will see, register liveness analysis can, in mnay case, provide the necessary registers at no cost.

Compared to traditional interprocedural variable liveness analysis, interprocedural register liveness analysis in executable code is both easier and harder. It is easier because there is no aliasing between registers and the number of registers for any given processor is bounded by a constant. It is harder because the total size of the control flow graphs is huge, given that the entire program (including libraries) is available at once. One has to be very careful to keep space and time requirements of the analysis reasonable.

This paper shows how to ompute register liveness information in a time and space efficient manner and how to cope with irregular control flow not encountered in high level languages. Tradeoffs between the precision of the analysis and its computation time and space are discussed. In particular, context sensitive and context insensitive register liveness analyses are compared. Experimental results on precision, time and space usage are provided as well as the impact of liveness analysis on optimizations. For the integer subset of the SPEC95 benchmark suite, we find that optimizations using the most accurate liveness analysis may reduce execution time by an additional 7% over the same optimizations using the trivial liveness analysis which assumes that all registers are live at the end of a node (basic block), even for programs compiled with a high degree of compiler optimization.

Related Work: Work most closely related to our own has been done by Srivastava and Wall on the OM optimizer [14] and by Goodwin on the Spike optimizer [2]. We improve on their liveness analysis in three ways. Firstly, we have changed the underlying flow equations resulting in three sets of almost identical equations, which simplifies implementation and reasoning about correctness. Secondly, we accelerate the fixpoint iteration by exploiting a novel insight about the interdependence of the various pieces of data flow information. This idea is also applicable to liveness analysis of variables. Thirdly, we show how to reduce the space requirement of the analysis by recomputation and exploitation of the new data flow equations.

We also show how to adapt liveness analysis to an environment where (assembly) code maybe handwritten and hence may not satisfy the “clean” control flow behavior of compiler generated code. Our approach, using so called compensation edges, is automatic while earlier proposals [16] special cased all the badly behaved functions in the standard libraries and hence could not cope with other misbehaved functions.

Furthermore, we explore ways to improve the accuracy of liveness analysis. For a known technique involving callee-save registers we point out a possible generalization.

Larus and Ball [6] describe a more ad hoc scheme to “scavenge” registers for instrumentation purposes. However, they fall back to spilling when programs are compiled with higher optimization levels or when code is hand tuned and therefore does not conform with calling conventions.

2 Basics

2.1 The Interprocedural Control Flow Graph

The basis for any kind of interprocedural data flow analysis is the interprocedural control flow graph (ICFG), or some equivalent representation of the program. The construction of such a control flow graph for ordinary programming languages is standard [4, 11]. For executables it is slightly more difficult but similar.

An ICFG consists of the (intraprocedural) control flow graphs (CFG) of the functions within the program, together with additional edges and nodes to account for interprocedural control flow. A call to a function f is modeled as depicted in Figure 1. The callsite (n_c, n_r) consists of two nodes the call node n_c and a return node n_r. There is a call edge from n_c to the init node of f, and a return edge from the exit node of f to n_r. The exit node is the unique successor node of all nodes that return from the function. Hence execution of a function starts at the init node and ends at the exit node.

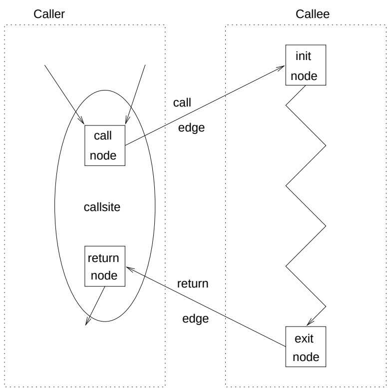

Figure 1: Modeling of Function Calls in an ICFG

The set of all nodes of the ICFG is denoted $Nodes$ and $Nodes[f]$ denotes the subset of nodes belonging to function f. The set of all functions is denoted $Functions$, the set of all edges is denoted $Edges$. The distinguished function $entryfun$ is where the execution of the program begins. For a node n, $Type[n]$ denotes its kind, ie. $call$, $return$, $init$, $exit$ or $other$. The set of immediate successor (resp. predecessor) nodes of a node n is denoted $Succ[n]$ (resp. $Pred[n]$). For any function $f \in Functions$, $InitNode[f]$ (resp. $ExitNode[f]$) denotes the init (resp. exit) node of f.

For any call node n_c, $ReturnNode[n_c]$ denotes the corresponding return node and $Callee[n_c]$ denotes the function being called. Similarly, for any return node n_r, $CallNode[n_r]$ denotes the corresponding call node and $Callee[n_r]$ denotes the function that was called.

2.2 Interprocedural Data Flow Analyses

Intraprocedural data flow analyses consider all possible paths in the CFG of a function to give an estimate of what data flow facts hold at a given node. Conditionals are not interpreted so this approach potentially includes paths that are never executed in reality and the estimate will be somewhat conservative.

For interprocedural data flow analyses we can simply adopt the intraprocedural approach and regard the interprocedural CFG as one big ordinary CFG, treating call and return edges as regular edges. Analyses performed in this fashion are called context insensitive interprocedural analyses. Such analyses are simple but often yield rather conservative estimates since many paths in the ICFG do not reflect real program executions. An example is shown in Figure 2 where two callsites call the same function f. Consider the path $C1 \rightarrow IN \rightarrow EX \rightarrow R2$. This path returns to the wrong callsite and hence does not occur in any execution. But since variable $s2$ is used in $R2$ and not defined along the path we conclude that $s2$ is live at $C1$, while in fact $s2$ is dead, as it is defined in $R1$.

Paths which do not return to the wrong callsite are called realizable paths, eg. $C1 \rightarrow IN \rightarrow EX \rightarrow R1$ or $C2 \rightarrow IN \rightarrow EX \rightarrow R2$. See [4] for a more rigorous definition.

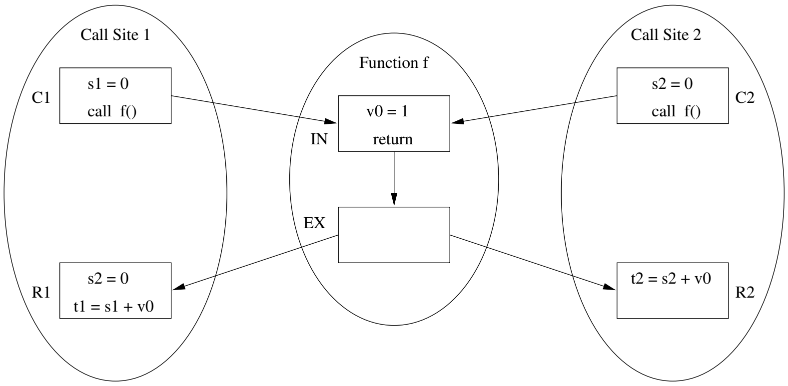

Figure 2: Imprecision

A context sensitive interprocedural data flow analysis considers only realizable paths in the ICFG [8].

3 Interprocedural Liveness Analysis

3.1 Context Insensitive Analysis

As described in the previous section, the context insensitive liveness analysis uses the standard intraprocedural analysis [9] and applies it to a program's ICFG treating call and return edges as ordinary edges.

The analysis iteratively computes the fixpoint of the equations below

$$\begin{array}{lll} LiveIn[n] & = use[n] \cup (LiveOut[n] - def[n]) & n \in Nodes \\ LiveOut[n] & = \bigcup s \in Succ[n] : LiveIn[s] & n \in Nodes \end{array}$$

subject to the initial values

$$\begin{array}{lll} LiveOut[n] & := \emptyset & n \in Nodes \\ LiveIn[n] & := \emptyset & n \in Nodes \end{array}$$

$\mathcal{R}$ denotes the set of all registers. For each node n, $def[n]$ contains the registers which are defined in n, $use[n]$ contains the registers which are used before they are defined in n.

3.2 Context Sensitive Analysis

For the context sensitive liveness analysis we must restrict ourselves to realizable paths through the ICFG. This is achieved by considering intraprocedural paths only and modeling function calls using summary information for the called function [8]. Conceptually, all call and return edges are removed from the ICFG and an edge between a call node n_c and its corresponding return node n_r is introduced. Data flow through that edge is subject to modifications described by the summary information for the called function.

Two pieces of information are necessary to summarize the effects of each function f on liveness:

- $MayUse[f]$. The set of registers that may be used by f. A register r may be used by f if there is a realizable path from $InitNode[f]$ to a use of r without an intervening definition of r. $MayUse[f]$ hence describes the set of registers which are always live at $InitNode[f]$ independent of the calling context and hence will be live at n_c. Typically these are the registers which are used to pass arguments to function f.
- $ByPass[f]$. The set of registers which if live at n_r will be live at n_c. Typically these are the register which are not used at all by f.

We also define

- $MustDef[f]$. The set of registers which must be defined (written to) on all paths from $InitNode[f]$ to $ExitNode[f]$.
- $MustDead[f]$. The the set of registers which must be defined on all paths from $InitNode[f]$ to $ExitNode[f]$ and are not used before they are defined. Clearly, $MustDead[f] = MustDef[f] - MayUse[f]$

The description of $ByPass[f]$ above does not define it uniquely. If a register is always live at the callsites calling f then we can include it in $ByPass[f]$ even if the register is never live at any of the corresponding return nodes. Consequently, we have a certain freedom of choice for $ByPass[f]$. Srivastava et al. [14] choose $ByPass[f]$ to be $\overline{MustDead}[f]$. The problem with this approach that it introduces a mutual dependency between *ByPass* information and *MayUse* information which complicates the flow equations. Goodwin [2] chooses $ByPass[f]$ to be $\overline{MustDef}[f]$ which does not have this problem and is therefore preferable. In fact, any set which lies between $\overline{MustDef}[f]$ and $\overline{MustDef}[f] \cup MayUse[f]$ is a valid candidate for $ByPass[f]$. Our choice for $ByPass[f]$ is a superset of Goodwin's [1] and will result in more uniform data flow equations.

Our analysis proceeds in three phases. The first two phases compute the summary information for functions, which is used by the third phase to perform the actual liveness computation.

Phase 1 : *Computation of byPass*$[f]$: iteratively compute the fixpoint of the data flow equations listed below

$$\begin{array}{lll}
ByPassIn[n] & = use[n] \cup (ByPassOut[n] - def[n]) & n \in Nodes \\
ByPassOut[n] & = \bigcup s \in Succ[n] : ByPassIn[s] & n \in Nodes \ \wedge \ Type[n] \notin \{call, exit\} \\
 & = \quad (ByPass[Callee[n]] \cap ByPassIn[n']) & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & & n' = ReturnNode[n] \\
ByPass[f] & = ByPassIn[InitNode[f]] & f \in Functions
\end{array}$$

Subject to the initial values

$$\begin{array}{lll}
ByPassOut[n] & := \emptyset & n \in Nodes \wedge \ Type[n] \neq exit \\
 & := \mathcal{R} & n \in Nodes \wedge \ Type[n] = exit \\
ByPassIn[n] & := \emptyset & n \in Nodes \\
ByPass[f] & := \emptyset & f \in Functions
\end{array}$$

Note that contrary to the intraprocedural liveness analysis or the context insensitive analysis the choice of the starting values is crucial, eg. initializing $byPassOut[f]$ of non exit nodes to $\mathcal{R}$ as in [2] yields overly conservative results [3].

Phase 2 : *Computation of mayUse*$[f]$: iteratively compute the fixpoint of the data flow equations listed below

$$\begin{array}{lll}
MayUseIn[n] & = use[n] \cup (MayUseOut[n] - def[n]) & n \in Nodes \\
MayUseOut[n] & = \bigcup s \in Succ[n] : MayUseIn[s] & n \in Nodes \ \wedge \ Type[n] \notin \{call, exit\} \\
 & = MayUse[f] \cup (ByPass[f] \cap MayUseIn[n']) & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & & n' = ReturnNode[n] \ \wedge \ f = Callee[n] \\
MayUse[f] & = MayUseIn[InitNode[f]] & f \in Functions
\end{array}$$

subject to the initial values

$$\begin{array}{lll}
MayUseOut[n] & := \emptyset & n \in Nodes \\
MayUseIn[n] & := \emptyset & n \in Nodes \\
MayUse[f] & := \emptyset & f \in Functions \\
ByPass[f] & := \text{as computed in Phase 1} & f \in Functions
\end{array}$$

Phase 3 : *Computation of liveness information*: iteratively compute the fixpoint of the data flow equations listed below

$$\begin{array}{lll}
LiveIn[n] & = use[n] \cup (LiveOut[n] - def[n]) & n \in Nodes \\
LiveOut[n] & = \bigcup s \in Succ[n] : LiveIn[s] & n \in Nodes \ \wedge \ Type[n] \notin \{call\} q \\
 & = MayUse[f] \cup (ByPass[f] \cap LiveIn[n']) & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & & n' = ReturnNode[n] \ \wedge \ f = Callee[n]
\end{array}$$

[1] it is difficult to give more intuitive description for this choice other than the fixpoint equations

Unified Dataflow Equations:

$$
\begin{array}{lll}
DataIn[n] & = use[n] \cup (DataOut[n] - def[n]) & n \in Nodes \\
DataOut[n] & = \bigcup s \in Succ[n] : DataIn[s] & n \in Nodes \ \wedge \ Type[n] \notin \text{NoPropTypes} \\
 & = MayUse[f] \cup (ByPass[f] \cap DataIn[ReturnNode[n]]) & n \in Nodes \ \wedge \ Type[n] = call \\
Summary[f] & = DataIn[InitNode[f]] & f \in Functions
\end{array}
$$

Unified Initial Values:

$$
\begin{array}{lll}
DataOut[n] & := \emptyset & n \in Nodes \wedge \ Type[n] \neq exit \\
 & := \text{ExitDataOutInit} & n \in Nodes \wedge \ Type[n] = exit \\
DataIn[n] & := \emptyset & n \in Nodes \\
Summary[f] & := \emptyset & f \in Functions
\end{array}
$$

Phase Adaptations:

	DataIn	DataOut	NoPropTypes	Summary	ExitDataOutInit
Phase 1	ByPassIn	ByPassOut	$\{call, exit\}$	ByPass	$\mathcal{R}$
Phase 2	MayUseIn	MayUseOut	$\{call, exit\}$	MayUse	$\emptyset$
Phase 3	LiveIn	LiveOut	$\{call\}$	—	$\emptyset$

Figure 3: Unified Fixpoint Computation

subject to the initial values

$$
\begin{array}{lll}
LiveOut[n] & := \emptyset & n \in Nodes \\
LiveIn[n] & := \emptyset & n \in Nodes \\
MayUse[f] & := \text{as computed in Phase 2} & f \in Functions \\
byPass[f] & := \text{as computed in Phase 1} & f \in Functions
\end{array}
$$

Note that phase 3 propagates liveness information into the exit nodes while phases 1and 2 do not.

Differing from Goodwin's approach we have modified the equation for $ByPassIn[n]$ by adding (unioning) $use[n]$ to the right hand side. This makes our *ByPass* sets strictly bigger than his but since $use[n] \subseteq MayUseIn[n]$ holds, $ByPassIn[InitNode[f]]$ will still lie between $\overline{MustDef}[f]$ and $\overline{MustDef}[f] \cup MayUse[f]$. The major virtue of this change is that it makes the equations of the three phases sufficiently similar that they can be unified into just one simple and compact set of equations (c.f. Figure 3). The code implementing the analysis, which uses the unified equations by means of a subroutine call is also correspondingly simpler and smaller. The bigger sets do not affect the performance if they are realized as bit vectors.

3.3 Tuning the Context Sensitive Analysis

The three phases above can be run in parallel. However, if executed sequentially the space used to hold $ByPassOut[n]$ and $ByPassIn[n]$ in Phase 1 can be used to hold $MayUseOut[n]$ and $MayUseIn[n]$ in Phase 2 which in turn can be reused in Phase 3 to hold $LiveIn[n]$ and $LiveOut[n]$. When comparing Phase 2 with Phase 3 it becomes evident that the fixpoint for $LiveOut[n]$ resp. $LiveIn[n]$ must be a superset of the fixpoint for $MayUseOut[n]$ resp. $MayUseIn[n]$. Hence, it is safe to initialize $LiveIn[n] := MayUseIn[n]$ and $LiveOut[n] := MayUseOut[n]$ thereby accelerating Phase 3 by not having to start the fixpoint iteration from scratch.

Next we describe how to improve Phase 3 more drastically exploiting the following observation. We focus on Out-sets here, In-sets are analogous. For a register r at node n of function f, we have

$$r \in LiveOut[n] \Rightarrow r \in MayUseOut[n] \ \vee \ r \in ByPassOut[n]$$

Conversely,

$$r \in MayUseOut[n] \Rightarrow r \in LiveOut[n]$$

But $r \in ByPassOut[n] \not\Rightarrow r \in LiveOut[n]$. The later does not hold because our initial values for *ByPassOut* of the exit nodes were pessimistic; we essentially assumed that all registers could be live. During Phase 3 it might turn out that not all registers are live at some exit nodes. The correct condition is therefore

$$r \in ByPassOut[n] \ \wedge \ r \in LiveOut[ExitNode[f]] \Rightarrow r \in LiveOut[n] \tag{1}$$

This suggests the following alternative approach for Phase 3 which has the virtue that it only iterates over the call graph rather than the much bigger supergraph.

```
 (1)  FOREACH n∈Nodes DO
 (2)    LiveOut[n]:= MayUseOut[n]
 (3)    LiveIn[n] := MayUseIn[n]
 (4)  REPEAT
 (5)    changed := false
 (6)    FOREACH f∈Functions DO
 (7)      new_out := ⋃ s∈Succ[ExitNode[f]] : LiveIn[s]
 (8)      IF new_out ≠ LiveOut[ExitNode[f]] THEN
 (9)        changed := true
(10)        liveOut[exit[f]] := new_out
(11)        FOREACH n∈Nodes[f] DO
(12)           LiveOut[n]:= MayUseOut[n]∪(ByPassOut[n] ∩ new_out)
(13)           LiveIn[n] := MayUseIn[n] ∪(ByPassIn[n] ∩ new_out)
(14)  UNTIL ¬changed
```

We begin by setting the start values for the fixpoint iterations using the improvement mentioned above (Lines 1 through 3). Then, we recompute the liveness information at the exit nodes for all functions until there is no change (Lines 4-14). If the liveness information at an exit node has changed we propagate this change according to (1) to all nodes of this function (Lines 11 through 13). Note, that it would suffice to propagate this information to return nodes only.

LiveOut and *MayUseOut* (resp. *LiveIn* and *MayUseIn*) need not be kept in separate locations; they can be merged into one, ie. all occurrences of *LiveOut* (resp. *LiveIn*) can be replaced by *MayUseOut* (resp. *MayUseIn*) which will then contain the liveness information upon completion of the fixpoint iteration. This also renders the first three lines of the algorithm unnecessary.

Since the last phase is usually the costliest of the three, this improvement cuts down execution time by 25%. (See the following section for experimental results). The drawback is that space usage almost doubles because both ByPass and MayUse information have to be kept around for each node (assuming *Live* information has been merged with *MayUse* information).

The enhancement is also applicable to ordinary interprocedural liveness analyses of variables.

3.4 Implementation and Performance of the Liveness Analyses

We have implemented the context sensitive and context insensitive liveness analysis algorithms as part of Alto [10], an optimizer for DEC UNIX/Alpha executables. Besides the speed of the analysis, space consumption was of primary concern to us. We found that it is usually better to recompute a data item than to store it. Thus, Alto only stores the various *Out*-sets associated with a node. The *In*-sets are computed by traversing the instructions of a basic block backwards. [2] def and *use* sets are not needed at all.

The relatively small number of instructions in a typical node make this approach viable. We also do not maintain a worklist of those nodes that need to be reconsidered during the fixed point iteration because this would incur the cost of another pointer per node. Instead, we mark those nodes which need recomputation and iterate over all nodes, processing marked nodes until no marked ones are left.

The total space requirement for the context insensitive liveness analysis is 64 bits per node to hold the *LiveOut* information. (1 bit for each of the 64 registers of the Alpha CPU). For the context sensitive analysis running the three phases sequentially we need an additional 128 bits per function to hold the *ByPass* and *MayUse* summary information simultaneously. For the improved version of the context sensitive analysis described in the previous section we need an additional 64 bits per node because we need to access *MayUseOut* and *ByPassOut* simultaneously.

[2] Alternatively, we could keep the *IN*-sets and recompute the *OUT*-sets from the successor nodes. However, when an optimization needs to determine which registers are live at a point within a node, it is more convenient to have the *OUT*-sets readily available.

Our experiments are based on the integer subset of the SPEC95 benchmark suite. Figure 4 summarizes their basic characteristics. The benchmark programs were statically compiled, hence the numbers include library functions.

Benchmark	Instructions	CFG Edges	Nodes	Functions
compress	18759	9222	5017	241
gcc	295096	158721	77501	2127
go	71721	29452	15692	602
ijpeg	54611	21530	11530	636
li	34768	17723	9134	643
m88ksim	46117	21938	11469	525
perl	90318	44995	22658	615
vortex	127383	58105	28461	1023

Figure 4: Characteristics of SPEC95 integer executables

Figure 5 shows our experimental results for the liveness analyses. Besides time and space usage we also measured the precision. For the improved context sensitive analysis, the space and time requirements are given in parentheses (the precision is not affected). The precision is computed as the average number of dead registers at the end of all nodes, ie. the average size of the $\overline{\mathit{LiveOut}}$ sets restricted to integer registers.

The last column contains the difference in precision between the context sensitive and insensitive analysis.

The measurements were obtained on a DEC Alpha workstation, with a 300MHz Alpha 21164 processor and 512MBytes of main memory, running DEC UNIX 4.0. The benchmarks were compile with the DEC C compiler V5.2-036 invoked as `cc -O4 -non_shared`. This instructs the C compiler to use the highest optimization level, with additional flags to make the linker retain relocation information, and results in statically linked executables.

Benchmark	Context Insensitive			Context Sensitive (improved)			Δ
	Space	Time	Prec.	Space	Time	Prec.	Prec.
compress	40kB	0.05s	2.9	44kB(84kB)	0.15s (0.10s)	4.1	1.2
gcc	620kB	1.30s	2.9	654kB(1274kB)	3.75s (3.00s)	5.5	2.6
go	126kB	0.20s	4.3	136kB(262kB)	0.55s (0.40s)	8.8	4.5
ijpeg	92kB	0.15s	2.9	102kB(194kB)	0.40s (0.30s)	3.8	0.9
li	73kB	0.10s	2.4	83kB(156kB)	0.30s (0.20s)	4.0	1.6
m88ksim	88kB	0.15s	3.0	96kB(184kB)	0.35s (0.250)	5.0	2.0
perl	181kB	0.30s	2.9	191kB(372kB)	0.85s (0.65s)	4.6	1.7
vortex	228kB	0.45s	2.9	244kB(472kB)	1.30s (1.00s)	5.5	2.6

Figure 5: Performance Liveness Analysis

The context sensitive analysis finds between 0.9 to 4.5 more dead register per node than the insensitive analysis and takes roughly three times as long to compute. Our improvement to the context sensitive analysis speeds the computation up by approximately 25% at the cost of a roughly twice the memory usage.

It is surprising that on the average at least 4 registers seem to be dead at the end of each node. Those dead registers could be used as scratch registers by an instrumentation tool such as pixie instead of spilling registers to memory thereby reducing profiling overhead.

4 Control Flow Irregularities

In this section we examine possible anomalies in the interprocedural control flow graph of executables. These anomalies are not encountered in flow graphs derived from high level languages. But ignoring them will result in an incorrect analysis.

4.1 Unknown Control Flow

Unknown control flow is rather common in executable code since much of the information present in the source code of the executable has been lost during compilation and linking. The three major sources for unknown control flow are

- switch or case statements compiled into computed jumps
  The compiler generates stylized instruction sequences for switch statements and hence by careful analysis the location and the size of the jump table can be recovered. If, due to optimizations or hand tuning, the stylized instruction sequence becomes obfuscated, we are unable to determine the possible successors of the jump node and and consequently are not able to determine how other nodes can be reached.

- exception handling
  This issue is, of course, very implementation specific but often quite similar to switch/case statements. When an exception is raised control is usually transferred via a computed jump to a handler (which might be in a different function). The possible targets of the jump are unknown and for a given handler the possible predecessor nodes are unknown.

- invocation through function pointers
  This can be either explicit or implicit (virtual functions), so that for some callsites we are unable to identify the function being called. Conversely, we are sometimes unable to identify all the callsites calling a given function.

The following standard technique [14] is used to cope with unknown control flow. It is safe but very crude. A special node $hellnode \in Nodes$ and a special function $hellfun \in Functions$ are added to the ICFG. If not all possible successor nodes of a node n can be determined, an extra edge from n to $hellnode$ is added. (If n is a call node then a call to $hellfun$ is added instead.) If not all possible predecessor nodes of a node n can be determined, an edge from $hellnode$ to n is added. (If n is an init node then a call from $hellfun$ is added to the function containing n instead; in particular $hellfun$ calls $entryfun$, the distinguished function where program execution begins.)

For the data flow analysis $hellnode$ and $hellfun$ will be associated with the worst case assumptions:

$$
\begin{array}{llll}
ByPassOut[hellnode] & := \mathcal{R} & LiveOut[hellnode] & := \mathcal{R} \\
ByPassIn[hellnode] & := \mathcal{R} & LiveIn[hellnode] & := \mathcal{R} \\
MayUseOut[hellnode] & := \mathcal{R} & MayUse[hellfun] & := \mathcal{R} \\
MayUseIn[hellnode] & := \mathcal{R} & ByPass[hellfun] & := \mathcal{R}
\end{array}
$$

For the context insensitive analysis we will identify $hellnode$ with $hellfun$.

4.2 Escaping Branches

Escaping branches are branches from the middle of a function into the middle of another. Traditional ICFGs [11, 4] do not consider edges corresponding to those branches. The only allowable interprocedural edges are call and and return edges. Escaping edges occur frequently in hand tuned assembly code, the UNIX system library of DEC UNIX is a good example. If an error occurs during a system call (eg. open, close, read, etc.) -1 is returned and the global variable `errno` is set to an special error code describing the error. The library designers have factored out the code that sets the `errno` variable into a function `doerror`. Figure 6 depicts the situation for the system function `close`.

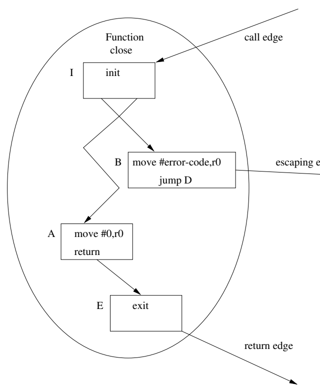

Figure 6: Escaping Edges

Since no function directly calls `doerror`, a naive analysis might find that register r0 is not live at the end of node *D*.The move instruction in *D* could then be deleted resulting in an incorrect program. Note, that this problem also needs to be addressed in a context insensitive analysis.

A simple fix would be to treat all functions with escaping edges as if we can not say anything about them, ie. similar to $hellfun$. We found this to be too limited because of the extra imprecision introduced by this approach. Our solution to the "escaping edge problem" is to introduce compensation edges into the ICFG. If there is an escaping branch from function f to g then a compensation edge from $ExitNode[g]$ to $ExitNode[f]$ is added. Interprocedural liveness analysis now works as usual because liveness information reaching $ExitNode[open]$ also propagates through the compensation edge to $ExitNode[doerror]$ and does not cause r0 to be dead. The situation is depicted in Figure 7.

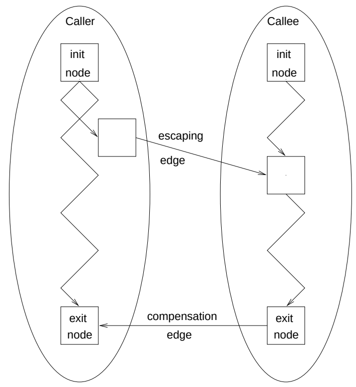

Figure 7: Compensation Edges

4.3 `setjmp` and `longjmp`

The standard C library contains two functions, `setjmp` and `longjmp`, which exhibit a rather irregular control flow behavior. Despite their frequent use, eg. in the three SPEC95 benchmarks gcc,li, and perl, they are neglected in the literature .

`setjmp` saves program state, such as registers and the program counter, into a memory area passed to it as an argument. If `longjmp` is called with the same memory area it will restore that program state. Hence, the return node of a callsite of `setjmp` can potentially be reached from many nodes while the return node of a callsite of `longjmp` will probably not be reached at all. Treating `setjmp` and `longjmp` as regular functions does not model the control flow correctly. The following scenario demonstrates this. Suppose a (global) variable is defined just before a call to `longjmp`. This definition might appear to be dead in a naive analysis because this variable might not be used (live) at the corresponding return node.

Using compensation edges, the problems can be solved quite simply, keeping the modifications to the ICFG at a bare minimum and introducing only a moderate amount of imprecision. For the function `setjmp` we add a compensation edge from *hellnode* to the *ExitNode*[*setjmp*]. For the function `longjmp` we add a compensation edge from *ExitNode*[*longjmp*] to *hellnode*. The situation is depicted in Figure 8.

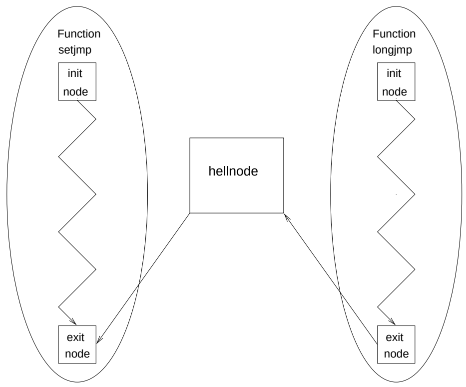

Figure 8: `Setjmp` and `longjmp`

5 Improving the Precision of Context Sensitive Register Liveness Analysis

This section explores on how the precision of liveness analysis can be improved. An obvious source for improvement is our overly pessimistic treatment of *hellnode* and *hellfun*. This will be exploited in 5.2. Section 5.1 shows how some registers which the analysis correctly identified as live can nevertheless be regarded as dead in some contexts.

5.1 Callee Save Registers

As described by Goodwin in [2], information about callee save registers can be exploited to reduce the number of live registers. Let $Saved[f]$ denoted the registers which are saved and restored by f and which are otherwise not used before defined in f. $Saved[f]$ will be a subset of $MayUse[f]$ because the saving of a register at function entry will be regarded as a use of that register by the liveness analysis. However, this use is only relevant if is the register is live at the return node of a given callsite.

Hence we can remove $Saved[f]$ from $MayUse[f]$ and instead add it to $ByPass[f]$ without affecting *safety*. The following slight modification of the equations updating the summary information in Phase 1 and 2 achieves the desired effect.

$$\begin{array}{lll} ByPass[f] & = ByPassIn[InitNode[f]] \cup Saved[f] & f \in Functions \\ MayUse[f] & = MayUseIn[InitNode[f]] - Saved[f] & f \in Functions \end{array}$$

In order to get a better insight how this optimization opportunity arises and how it may be generalized we consider the following (hypothetical) code for complex addition and two of its callsites.

```
                          Callsite1:
                              ...
ComplexAdd:                   load r10,memloc1          Callsite2:
    add r10,r12,r0            load r11,memloc2              ...
    add r11,r13,r1            load r12,memloc3              bsr  ComplexAdd
```

```
    ret                        load r13,memloc4         mul  r0,r0,r0
                               bsr  ComplexAdd          mul  r1,r1,r1
                               move r0,r10              add  r0,r1,r0
                               bsr  PrintNumber         ...
                               ...
```

For `ComplexAdd` the real and imaginary part of the first summand is passed in registers r10 and r11, the real and imaginary part of the second summand in registers r12 and r13, and the result is returned in r0 and r1. `Callsite1` just prints out the real part of the result, while `Callsite2` computes its squared norm.

Clearly, registers r10 through r13 will be live at both callsites just before the call to `ComplexAdd`. But `Callsite1` only uses the real part of the result hence the result computed by the second add in `ComplexAdd` is useless. A lazy programming language would neither execute this add instruction nor the instructions computing the values of registers r11 and r13. Unfortunately, we cannot eliminate the add instruction since `Callsite2` uses both results r0 and r1. Be we can consider registers r11 and r13 to be dead at `Callsite1` and subsequently eliminate the corresponding load instructions. Registers r11 and r13 will then have arbitrary values and the add instruction produces an arbitrary result which is ignored [3]

In this light the callee save register s can be regarded as an additional argument a_s and result r_s of the function f. (s, a_s, and r_s will of course denote the same register.) a_s will be moved to a new location and then from there to r_s. If r_s is not live at a given returnnode the move operations are useless. But as above we cannot delete them. All we can do is mark a_s as dead at the corresponding callnode and this is exactly what is achieved by moving s from $MayUse[f]$ to $ByPass[f]$.

5.2 Calling Conventions

Suppose function f does not use or define register r and does not call any other function. Our liveness analysis will determine that $r \in ByPass[f]$. Now assume that f and any function calling f obey some sort of calling convention which state that register r is not preserved across procedure calls and does not carry a result. This implies that r will not be live at any return node of a callsite of f and it is therefore safe to remove r from any $ByPass[f]$. In fact, it is irrelevant whether r is in $ByPass[f]$ or not. The smaller $ByPass$ set is nevertheless desirable, because $hellfun$ or $hellnode$ may introduce unwanted liveness information into the analysis which would be partially eliminated by the smaller set. Unfortunately, we have no control over the enforcement of calling conventions in general, except for system calls. In fact, compilers often violate calling convention when they perform interprocedural register allocation or when library functions are invoked that implement missing hardware features such as a divide instruction. It seems reasonable, however, to assume that calls to shared libraries and calls through function pointers respect the calling convention.

In our current version of the liveness analysis those calls are modeled by a call to $hellfun$. An enhancement would be to model this as a call to a different function $sysfun$ (or a special node $sysnode$ for a context insensitive analysis). [4]

Let $sysuse$ denote the set of registers potentially used according to the calling conventions and $syssave$ the set of registers preserved across function calls. The liveness analysis will be augmented with the following assignments.

$$
\begin{array}{llll}
ByPassOut[sysnode] & := syssave & LiveOut[sysnode] & := sysuse \cup syssave \\
ByPassIn[sysnode] & := syssave & LiveIn[sysnode] & := sysuse \cup syssave \\
MayUseOut[sysnode] & := sysuse & MayUse[sysfun] & := sysuse \\
MayUseIn[sysnode] & := sysuse & ByPass[sysfun] & := syssave
\end{array}
$$

[3] If add can cause a side effect such as an overflow this approach is of course not valid.

[4] If the calling conventions for system calls differ from those for regular functions, as it is the case for DEC UNIX for the Alpha, we introduce one function/node for each set of calling conventions

5.3 Performance

We have added the enhancements described in the previous sections to the context sensitive analysis and measured the resulting gain in precision. Figure 9 shows the average number of dead registers at the end of a node without any enhancement, with one of the enhancements, and with both enhancements.

Our experiments show that incorporating both enhancements increases the number of dead registers by between 1.8 and 4.3. The enhancements seem to be synergetic since the total improvement is bigger than the sum of the individual improvements.

Benchmark	None	Save	Call. Conv.	Both	Both - None
compress	4.1	4.2	6.8	7.6	3.5
gcc	5.5	6.0	7.4	8.5	3.0
go	8.8	9.0	10.2	10.6	1.8
ijpeg	3.8	3.9	7.6	8.1	4.3
li	4.0	4.2	6.1	6.6	2.6
m88ksim	5.0	5.3	7.1	8.0	3.0
perl	4.6	4.9	7.1	8.0	3.4
vortex	5.5	6.2	7.2	8.7	2.2

Figure 9: Impact of Enhancements to Liveness Analysis

6 Applying Register Liveness Information

Ultimately, the utility of an analysis should be measured by the extent to which it enables optimizations to be carried out. In particular, an analysis that attains improved precision at the cost of increased complexity should be justified by the additional code optimizations that become possible as a result of the improvement in precision.

There are a variety of optimizations that will benefit from register liveness information: (partially) dead code elimination, common subexpression elimination, loop invariant optimizations, moving stackvariables into registers, etc. In this section, we will focus on reload avoidance. In the next section, we evaluate the extent to which this and other optimizations are affected by the different versions of liveness analysis.

Reload avoidance attempts to reduced the number of load instructions in a program. If a value has been loaded from or stored to a memory location, then a subsequent load from the same location can be avoided if there are no intervening store instructions overwriting the location. Often, the reloaded value is still available in some register and the load instruction can be replaced by a move instruction (from that register). If the reloaded value is not available, liveness analysis might allow us to find a free scratch register that can carry it. Copy propagation can then be applied to eliminate those move instructions. But even if the move instructions cannot be eliminated there is a benefit, since move instructions typically have smaller latencies and fewer scheduling restrictions than load instructions. Our reload avoidance is performed on extended basic blocks.

6.1 Experimental Results

In order to measure the effectiveness of the liveness analysis, we optimized the integer subset of the SPEC95 benchmark suite with the Alto system. The benchmarks were compiled as described in Section 3.4. Profiling information was obtained using pixie and the training data sets provided by SPEC. (The profiling information is used by some of optimizations performed by Alto.) Actual measurements were done with the reference data sets.

Figure 10 shows the execution time of the optimized executables as reported by the UNIX time command (user time) while the system was very lightly loaded. This reflects the overall effectiveness of the optimizations performed by Alto. With the enhanced context sensitive liveness analysis, Alto is able to reduce the execution time by up to 7%

Benchmark	trivial	context insensitive	enhanced context sensitive
compress	261.2 (100.0%)	260.2 (99.6%)	261.2 (100.0%)
gcc	236.6 (100.0%)	231.1 (97.7%)	221.6 (93.7%)
go	344.5 (100.0%)	344.5 (100.0%)	336.7 (97.7%)
ijpeg	329.2 (100.0%)	325.5 (98.9%)	324.6 (98.6%)
li	254.1 (100.0%)	250.5 (98.6%)	245.9 (96.8%)
m88ksim	215.2 (100.0%)	219.1 (101.8%)	213.5 (99.2%)
perl	205.1 (100.0%)	200.6 (97.8%)	190.0 (92.6%)
vortex	346.6 (100.0%)	322.8 (93.1%)	321.5 (92.8%)

Figure 10: Running time in seconds after optimization with different liveness analyses

Benchmark	trivial	context insensitive	enhanced context sensitive
compress	12037M (100.0%)	11750M (97.6%)	11748M (97.6%)
gcc	11493M (100.0%)	11054M (96.2%)	10756M (93.6%)
go	19370M (100.0%)	18483M (95.4%)	17754M (91.7%)
ijpeg	20115M (100.0%)	19829M (98.5%)	19820M (98.5%)
li	17382M (100.0%)	16700M (96.0%)	16323M (93.9%)
m88ksim	15133M (100.0%)	14057M (92.9%)	13790M (91.1%)
perl	12311M (100.0%)	11684M (94.9%)	11241M (91.3%)
vortex	24183M (100.0%)	22916M (94.8%)	22408M (92.7%)

Figure 11: Dynamic loads after optimization with different liveness analyses

(vortex,perl) over the trivial liveness analysis which assumes that all registers are live at the end of a basic block. For benchmarks like compress and m88ksim, whose execution time is dominated by small loops, the improved liveness information has little or no benefit. In fact, compress and m88ksim exhibit an anomaly in the execution time for the context insensitive analysis which is probably due to deficiencies in our instruction ischeduler.

Figure 11 shows the (dynamic) number of load instructions executed by the optimized executables. [5] This primarily reflects the effectiveness of the load avoidance optimization. Compared with trivial liveness analysis, the enhanced context sensitive analysis will reduce the number of load instructions by up to 9%.

References

[1] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for Alpha/NT executables. In *The USENIX Windows NT Workshop*, Seattle, Washington,USA, August 1997.

[2] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *ACM '97 Conference on Programming Language Design and Implementation*, pages 122–133, Las Vegas, Nevada, USA, June 1997.

[3] David W. Goodwin. Personal communication, 1997.

[4] William Landi and Barbara Ryder. Pointer-induced aliasing: A problem taxonomy. In *ACM '91 Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, USA, January 1991.

[5] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, May 1993.

[6] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software — Practice and Experience*, 24(2):197–218, February 1994.

[5] The count where obtained using the hardware performance counter of the Alpha CPU

[7] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *ACM '95 Conference on Programming Language Design and Implementation*, New York, NY, USA, June 1995.

[8] Thomas J. Marlowe, Barbara Ryder, and Michael Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Rutgers University, July 1995.

[9] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[10] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. Alto: A link-time optimizer for the dec alpha. Technical Report ,http://www.cs.arizona.edu/people/debray/papers/alto.ps, Department of Computer Science,The University of Arizona, September 1998.

[11] Eugene W. Myers. A precise interprocedural data flow algorithm. In *ACM '81 Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[12] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *The USENIX Windows NT Workshop 1997*, Seattle, WA, USA, August 1997.

[13] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *ACM '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[14] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[15] MIPS Computer Systems. Riscompiler and c programmer's guide, 1989.

[16] David W. Wall. Systems for late code modification. Technical Report 92/3, Digital Equipment Corporation, Western Research Lab, May 1992.