Register Allocation Using Lazy Saves, Eager Restores, and Greedy Shuffling

Robert G. Burger Oscar Waddell R. Kent Dybvig

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{*burger,owaddell,dyb*}@*cs.indiana.edu*

Abstract

This paper presents a fast and effective linear intraprocedural register allocation strategy that optimizes register usage across procedure calls. It capitalizes on our observation that while procedures that do not contain calls (*syntactic leaf routines*) account for under one third of all procedure activations, procedures that actually make no calls (*effective leaf routines*) account for over two thirds of all procedure activations. Well-suited for both caller- and callee-save registers, our strategy employs a "lazy" save mechanism that avoids saves for all effective leaf routines, an "eager" restore mechanism that reduces the effect of memory latency, and a "greedy" register shuffling algorithm that does a remarkably good job of minimizing the need for temporaries in setting up procedure calls.

1 Introduction

Register allocation, the complex problem of deciding which values will be held in which registers over what portions of the program, encompasses several interrelated sub-problems. Perhaps the most well-known of these is to decide which variables to assign to registers so that there is no conflict [5]. Another involves splitting live ranges of variables in order to reduce conflicts. These problems have been addressed for both intraprocedural and interprocedural register allocation. Optimizing register usage across procedure calls is also an important problem, but up to now it has been addressed primarily in terms of interprocedural analysis.

In this paper we describe a fast and effective linear intraprocedural register allocation strategy that optimizes register usage across procedure calls. In conjunction with local register allocation, this strategy results in performance within range of that achieved by interprocedural register allocation. Furthermore, it is successful even in the presence of anonymous procedure calls, which typically cause interprocedural register allocation techniques to break down.

This material is based in part upon work supported under National Science Foundation Grant CDA-9312614 and a NSF Graduate Research Fellowship.

To appear in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation.*

Our compiler dedicates a set of registers to be used to hold procedure arguments, including the actual parameters and return address. Any unused registers (including registers containing non-live argument values) are available for intraprocedural allocation, both for user variables and compiler temporaries. Other registers are used for local register allocation and to hold global quantities such as the stack pointer and allocation pointer.

Three costs must be minimized in order to optimize register usage across procedure calls: the cost of saving live registers around calls, the cost of restoring saved registers before they are used, and the cost of "shuffling" argument registers when setting up the arguments to a call, some of which may depend upon the old argument values. It is easy for these costs to exceed the benefits of register allocation, especially with many registers dedicated to procedure arguments.

We analyzed the run-time call behavior for a large variety of Scheme programs. While procedures that contain no calls (*syntactic leaf routines*) account for under one third of all procedure activations, procedures that actually make no calls (*effective leaf routines*) account for over two thirds. We capitalize on this fact by using a "lazy" save mechanism that avoids saves for all effective leaf routines. In addition, we reduce the effect of memory latency by employing an "eager" restore mechanism. Although this approach sometimes introduces unnecessary restores, we found that a lazy mechanism did not improve performance and added considerable compile-time overhead. Finally, we employ a "greedy" register shuffling algorithm that does a remarkably good job of minimizing the need for temporaries in setting up procedure calls.

Section 2 presents our lazy save, eager restore, and greedy shuffling algorithms. Section 3 describes our implementation of the algorithms. Section 4 discusses the performance characteristics of our implementation. Section 5 describes related work. Section 6 summarizes our results and discusses possibilities for future work.

2 Save and Restore Placement

For purposes of discussion we describe our strategy for save and restore placement in terms of caller-save registers and the simplified language of expressions based on Scheme [8] below. In Section 2.4 we explain how our strategy applies to callee-save registers.

Benchmark	Lines	Description
Compiler	30,000	*Chez Scheme* recompiling itself
DDD	15,000	hardware derivation system [3] deriving Scheme machine [4]
Similix	7,000	self-application of the Similix [2] partial evaluator
SoftScheme	10,000	Wright's soft typer [15] checking a 2,500 line program

Table 1.

$$\begin{aligned} E &\rightarrow x \\ &\rightarrow \mathsf{true} \\ &\rightarrow \mathsf{false} \\ &\rightarrow \textbf{call} \\ &\rightarrow (\textbf{seq}\ E_1\ E_2) \\ &\rightarrow (\textbf{if}\ E_1\ E_2\ E_3) \end{aligned}$$

We assume that assignment conversion has already been done, so there are no assignment expressions. All constants are reduced to either true or false. For simplicity, we ignore the operator and operands of procedure calls by assuming they have been ordered in some way and placed in a series of **seq** expressions whose last entry is **call**.

2.1 Lazy Save Placement

The natural strategy for save placement involves two extremes: the callee-save registers used in a procedure are saved on entry, whereas the caller-save registers live after a call are saved right before the call. The natural callee-save strategy saves too soon in the sense that it may introduce unnecessary saves when a path through the procedure does not use the registers. The natural caller-save strategy saves too late in the sense that it may introduce redundant saves when a path contains multiple calls. Our unified strategy optimizes the placement of registers saves between these two extremes for both callee- and caller-save registers.

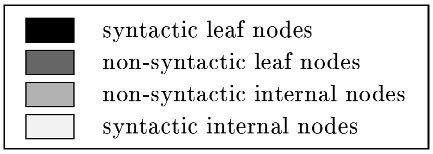
We tooled the *Chez Scheme* compiler to insert code to count procedure activations in a variety of programs and found that syntactic leaf routines (those that contain no calls[1]) on average account for under one third of all activations. We then retooled the compiler to determine how many activations actually make no calls. These *effective leaf routines* account for an average of more than two thirds of procedure activations. Our lazy save strategy thus caters to effective leaf routines, saving registers only when a call is inevitable. Because of assignment conversion, variables need to be saved only once. In order to minimize redundant saves, therefore, our strategy saves registers as soon as a call is inevitable. Table 2 gives the results of our measurements for a set of benchmarks described in Table 1 and for a Scheme version of the Gabriel benchmark suite [10]. Effective leaf routines are classified as syntactic and non-syntactic leaf nodes. Non-syntactic internal nodes are activations of procedures that have paths without calls but make calls at run time, and syntactic internal nodes are those that have no paths without calls.

First we present a simple algorithm for determining lazy save placement. Next we demonstrate a deficiency involving short-circuit boolean operations within **if** test expressions.

[1] Because tail calls in Scheme are essentially jumps, they are not considered calls for this purpose.

Benchmark	Calls	Breakdown
Compiler	33,041,034	
DDD	86,970,102	
Similix	33,891,834	
SoftScheme	11,153,705	
boyer	914,113	
browse	1,608,975	
cpstak	159,135	
ctak	302,621	
dderiv	191,219	
destruct	236,412	
div-iter	1,623	
div-rec	140,738	
fft	677,886	
fprint	43,715	
fread	23,194	
fxtak	63,673	
fxtriang	5,817,460	
puzzle	912,245	
tak	111,379	
takl	537,205	
takr	111,380	
tprint	41,940	
traverse-init	1,268,249	
traverse	7,784,102	
triang	11,790,492	
Average		

Table 2. Dynamic call graph summary



We then present an improved algorithm that handles these cases.

2.1.1 A Simple Save Placement Algorithm

We first define the function $\mathcal{S}[E]$, the set of registers that should be saved around expression E, recursively on the structure of our simplified expressions:

$$\begin{aligned} \mathcal{S}[x] &= \emptyset \\ \mathcal{S}[\mathsf{true}] &= \emptyset \\ \mathcal{S}[\mathsf{false}] &= \emptyset \\ \mathcal{S}[\textbf{call}] &= \{r \mid r \text{ is live after the call}\} \\ \mathcal{S}[(\textbf{seq}\ E_1\ E_2)] &= \mathcal{S}[E_1] \cup \mathcal{S}[E_2] \\ \mathcal{S}[(\textbf{if}\ E_1\ E_2\ E_3)] &= \mathcal{S}[E_1] \cup (\mathcal{S}[E_2] \cap \mathcal{S}[E_3]) \end{aligned}$$

We save register r around expression E iff. $r \in \mathcal{S}[E]$. By intersecting $\mathcal{S}[E_2]$ with $\mathcal{S}[E_3]$ in the **if** case, only those registers that need to be saved in both branches are propagated, which yields a lazy save placement. The union operator in the **seq** case places the saves as soon as they are inevitable. It can be shown that this placement is never too eager; *i.e.*, if there is a path through any expression E without making a call, then $\mathcal{S}[E] = \emptyset$.

2.1.2 Short-Circuit Boolean Expressions

Short-circuit boolean operations such as **and** and **or** are modeled as **if** expressions. As a result, **if** expressions nested in the "test" part occur frequently. Consider the expression (**if** (**and** x **call**) y **call**), which is modeled by (**if** (**if** x **call** false) y **call**). There is no path through this expression without making a call, so we would like to save all the live variables around the outer **if** expression. Unfortunately, the above algorithm is too lazy and would save none of the registers, regardless of which registers are live after the calls:

$$\begin{aligned}
&\mathcal{S}[(\textbf{if}\ (\textbf{if}\ x\ \textbf{call}\ \mathsf{false})\ y\ \textbf{call})] \\
&\quad = \mathcal{S}[(\textbf{if}\ x\ \textbf{call}\ \mathsf{false})] \cup (\mathcal{S}[y] \cap \mathcal{S}[\textbf{call}]) \\
&\quad = (\mathcal{S}[x] \cup (\mathcal{S}[\textbf{call}] \cap \mathcal{S}[\mathsf{false}])) \cup (\emptyset \cap \mathcal{S}[\textbf{call}]) \\
&\quad = \emptyset \cup (\mathcal{S}[\textbf{call}] \cap \emptyset) \cup \emptyset \\
&\quad = \emptyset
\end{aligned}$$

To correct this deficiency, we must be sensitive to conditionals and constants, especially when they occur in a test context.

2.1.3 The Revised Save Placement Algorithm

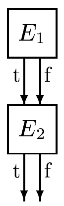
The basic principle is to consider the paths through the control flow graph. Along a given path we take the union of the registers that need to be saved at each node. Then we take the intersection of all the paths to determine the registers that need to be saved regardless of the path taken. In order to facilitate this process, we define two functions recursively: $\mathcal{S}_t[E]$, the set of registers to save around E if E should evaluate to true, and $\mathcal{S}_f[E]$, the set of registers to save around E if E should evaluate to false. Register r is saved around E iff. $r \in \mathcal{S}_t[E] \cap \mathcal{S}_f[E]$.

The base cases are defined as follows, where R is the set of all registers:

$$\begin{aligned}
\mathcal{S}_t[x] &= \emptyset & \mathcal{S}_f[x] &= \emptyset \\
\mathcal{S}_t[\mathsf{true}] &= \emptyset & \mathcal{S}_f[\mathsf{true}] &= R \\
\mathcal{S}_t[\mathsf{false}] &= R & \mathcal{S}_f[\mathsf{false}] &= \emptyset \\
\mathcal{S}_t[\textbf{call}] &= \{r \mid r \text{ is live after the call}\} & \mathcal{S}_f[\textbf{call}] &= \{r \mid r \text{ is live after the call}\}
\end{aligned}$$

Since it is impossible that true should evaluate to false, and *vice versa*, we define these cases to be R so that any impossible path will have a save set of R, the identity for intersection. Thus, impossible paths will not unnecessarily restrict the result.

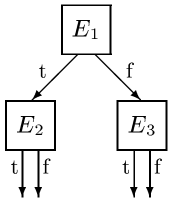
Now we define the recursive cases. Intuitively, the set $\mathcal{S}_t[(\textbf{seq}\ E_1\ E_2)]$ is the set of registers to save if the **seq** expression evaluates to true. There are two possible paths: E_1 is true and E_2 is true, or E_1 is false and E_2 is true. Thus, $\mathcal{S}_t[\textbf{seq}] = (\mathcal{S}_t[E_1] \cup \mathcal{S}_t[E_2]) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_t[E_2]) = (\mathcal{S}_t[E_1] \cap \mathcal{S}_f[E_1]) \cup \mathcal{S}_t[E_2]$. The case for $\mathcal{S}_f[(\textbf{seq}\ E_1\ E_2)]$ is similar, as the diagram illustrates:



(**seq** E_1 E_2)

$$\begin{aligned}
\mathcal{S}_t[\textbf{seq}] &= (\mathcal{S}_t[E_1] \cap \mathcal{S}_f[E_1]) \cup \mathcal{S}_t[E_2] \\
\mathcal{S}_f[\textbf{seq}] &= (\mathcal{S}_t[E_1] \cap \mathcal{S}_f[E_1]) \cup \mathcal{S}_f[E_2]
\end{aligned}$$

Next, consider the two paths for which (**if** E_1 E_2 E_3) evaluates to true: E_1 is true and E_2 is true, or E_1 is false and E_3 is true. Similarly, there are two paths for false, as the diagram illustrates:



(**if** E_1 E_2 E_3)

$$\begin{aligned}
\mathcal{S}_t[\textbf{if}] &= (\mathcal{S}_t[E_1] \cup \mathcal{S}_t[E_2]) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_t[E_3]) \\
\mathcal{S}_f[\textbf{if}] &= (\mathcal{S}_t[E_1] \cup \mathcal{S}_f[E_2]) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_f[E_3])
\end{aligned}$$

Our example $A =$ (**if** (**if** x **call** false) y **call**) now yields the desired result. Let L be the set of live registers after A. Let $B =$ (**if** x **call** false).

$$\begin{aligned}
\mathcal{S}_t[B] &= (\emptyset \cup (\{y\} \cup L)) \cap (\emptyset \cup R) \\
&= \{y\} \cup L
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_t[A] &= (\mathcal{S}_t[B] \cup \emptyset) \cap (\mathcal{S}_f[B] \cup L) \\
&= L \\
\mathcal{S}_f[B] &= (\emptyset \cup (\{y\} \cup L)) \cap (\emptyset \cup \emptyset) \\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_f[A] &= (\mathcal{S}_t[B] \cup \emptyset) \cap (\mathcal{S}_f[B] \cup L) \\
&= L
\end{aligned}$$

We see that although no registers would be saved around the inner **if** expression (since $\mathcal{S}_t[B] \cap \mathcal{S}_f[B] = \emptyset$), all the live registers would be saved around the outer **if** as desired.

It is straightforward to show that the revised algorithm is not as lazy as the previous algorithm, *i.e.*, that $\mathcal{S}[E] \subseteq \mathcal{S}_t[E] \cap \mathcal{S}_f[E]$ for all expressions E. It can also be shown that the revised algorithm is never too eager; *i.e.*, if there is a path through any expression E without calls, then $\mathcal{S}_t[E] \cap \mathcal{S}_f[E] = \emptyset$.

Figure 1 shows the control graphs for **not** and the short-circuit boolean operators **and** and **or** and the derived equations for these operators.

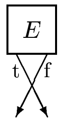
2.2 Eager Restore Placement

We considered two restore strategies based on the question of how soon a register r should be restored:

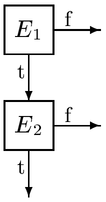
- eager: as soon as r *might* be needed, *i.e.*, if r will possibly be referenced before the next call, and
- lazy: as soon as r *will* be needed, *i.e.*, if r will certainly be referenced before the next call.

We present three abbreviated control flow diagrams to demonstrate the differences in the two approaches. The register save regions are indicated by a rounded box. Calls are indicated by squares, references by circles, and restores by dashes across the control flow lines. Control enters from the top.

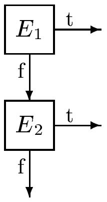
Figures 2a and 2b demonstrate how the eager approach introduces potentially unnecessary restores because of the joins of two branches with different call and reference behavior. Figure 2c shows an instance where even the lazy approach may be forced to make a potentially unnecessary restore. Because the variable is referenced outside of its enclosing save region, there is a path that does not save the variable. Consequently, the register must be restored on exit of the save region.



$$\begin{aligned}
\mathcal{S}_t[(\textbf{not}\ E)] &= \mathcal{S}_t[(\textbf{if}\ E\ \mathsf{false}\ \mathsf{true})] \\
&= (\mathcal{S}_t[E] \cup R) \cap (\mathcal{S}_f[E_1] \cup \emptyset) \\
&= \mathcal{S}_f[E] \\
\mathcal{S}_f[(\textbf{not}\ E)] &= \mathcal{S}_f[(\textbf{if}\ E\ \mathsf{false}\ \mathsf{true})] \\
&= (\mathcal{S}_t[E] \cup \emptyset) \cap (\mathcal{S}_f[E] \cup R) \\
&= \mathcal{S}_t[E]
\end{aligned}$$

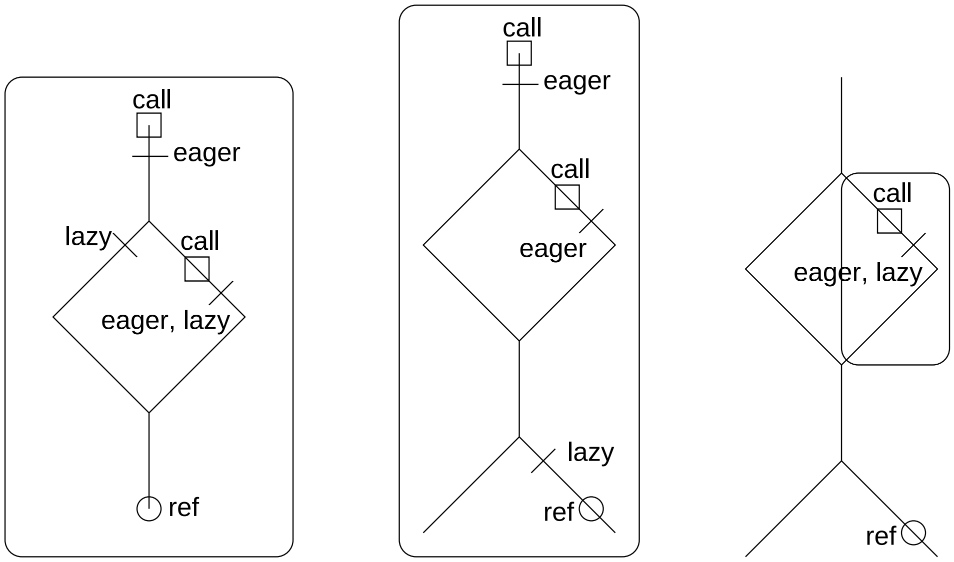


$$\begin{aligned}
\mathcal{S}_t[(\textbf{and}\ E_1\ E_2)] &= \mathcal{S}_t[(\textbf{if}\ E_1\ E_2\ \mathsf{false})] \\
&= (\mathcal{S}_t[E_1] \cup \mathcal{S}_t[E_2]) \cap (\mathcal{S}_f[E_1] \cup R) \\
&= \mathcal{S}_t[E_1] \cup \mathcal{S}_t[E_2] \\
\mathcal{S}_f[(\textbf{and}\ E_1\ E_2)] &= \mathcal{S}_f[(\textbf{if}\ E_1\ E_2\ \mathsf{false})] \\
&= (\mathcal{S}_t[E_1] \cup \mathcal{S}_f[E_2]) \cap (\mathcal{S}_f[E_1] \cup \emptyset) \\
&= (\mathcal{S}_t[E_1] \cup \mathcal{S}_f[E_2]) \cap \mathcal{S}_f[E_1]
\end{aligned}$$



$$\begin{aligned}
\mathcal{S}_t[(\textbf{or}\ E_1\ E_2)] &= \mathcal{S}_t[(\textbf{if}\ E_1\ \mathsf{true}\ E_2)] \\
&= (\mathcal{S}_t[E_1] \cup \emptyset) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_t[E_2]) \\
&= \mathcal{S}_t[E_1] \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_t[E_2]) \\
\mathcal{S}_f[(\textbf{or}\ E_1\ E_2)] &= \mathcal{S}_f[(\textbf{if}\ E_1\ \mathsf{true}\ E_2)] \\
&= (\mathcal{S}_t[E_1] \cup R) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_f[E_2]) \\
&= \mathcal{S}_f[E_1] \cup \mathcal{S}_f[E_2]
\end{aligned}$$

Figure 1.



a. **b.** **c.**

Figure 2.

In summary, the eager approach would immediately restore any register possibly referenced before the next call. It is straightforward and can be easily implemented in a single linear pass through the abstract syntax tree. A bottom-up pass through the tree could compute the variable reference information and insert the necessary restores in parallel. Because the restores occur early, they may reduce the effect of memory latency.

The lazy approach would restore whenever a reference is inevitable before the next call or when the register is live on exit from the enclosing save region. Its main advantage is that most unnecessary restores can be avoided. Unfortunately, this approach is less straightforward and requires more passes through the abstract syntax tree.

We implemented both approaches early on and found that the eager approach produced code that ran just as fast as the code produced by the lazy approach. We concluded from this that the reduced effect of memory latency offsets the cost of unnecessary restores.

2.3 Greedy Shuffling

When setting up arguments to a call, the new argument values may depend on the old argument values; thus, argument register shuffling is sometimes necessary. For example, consider the call $f(y, x)$, where at the time of the call x is in argument register a_1 and y in a_2. In order to call f, y must be assigned to a_1 and x to a_2, requiring a swap of these two argument registers.

By not fixing the order of evaluation for arguments before register allocation, the compiler can determine an ordering that requires a minimal amount of shuffling, and sometimes it can avoid shuffling altogether. For example, the call $f(x+y, y+1, y+z)$, where x is in register a_1, y is in register a_2, and z is in register a_3, can be set up without shuffling by evaluating $y+1$ last. A left-to-right or right-to-left ordering, however, would require a temporary location for this argument.

Because we do not fix the order of evaluation of arguments early on, we cannot compute liveness information in a pass before the shuffling. Since register allocation depends on liveness information, we must perform register allocation, shuffling, and live analysis in parallel.

Our shuffling algorithm first partitions the arguments into those that contain calls (*complex*) and those that do not (*simple*). We use temporary stack locations for all but the last of the complex arguments, since making a call would cause the previous arguments to be saved on the stack anyway. We pick as the last complex argument one on which none of the simple arguments depend (if such a complex argument exists), since it can be evaluated directly into its argument register.

Ordering the simple arguments is a problem of optimizing parallel assignments, as noted in [9, 12]. We build the dependency graph and essentially perform a topological sort. If we detect a cycle, we find the argument causing the largest number of dependencies and remove it, placing it into a temporary location, in hopes of breaking the cycle. We then continue the process until all arguments have been processed. This "greedy" approach to breaking cycles may not always result in the minimal number of temporaries. We have observed that our algorithm, however, finds the optimal ordering for a vast majority of call sites (see Section 3.1).

2.4 Callee-Save Registers

We now describe how our lazy save strategy applies to callee-save registers. As we noted earlier, the natural register save placement strategy of saving on entry all callee-save registers used in a procedure introduces unnecessary saves when a path through the procedure does not use the registers. Our effective leaf routine measurements indicate that paths without calls are executed frequently. Along these paths caller-save registers can be used without incurring any save/restore overhead. Using caller-save registers along paths without calls, our approach delays the use of callee-save registers to paths where a call is inevitable.

By adding a special caller-save return-address register, *ret*, the revised save placement algorithm can be used to determine which expressions will always generate a call. This return register is used to hold the return address of the current procedure and must be saved and restored around calls. Consequently, if $ret \in \mathcal{S}_t[E] \cap \mathcal{S}_f[E]$,[2] then E will inevitably make a call, but if $ret \notin \mathcal{S}_t[E] \cap \mathcal{S}_f[E]$, then E contains a path without any calls.

Chow [6] describes a related technique called "shrink wrapping" to move the saves and restores of callee-save registers to regions where the registers are active. His technique, however, is applied after variables have been assigned to callee-save registers. Consequently, his approach introduces unnecessary saves and restores when it assigns a variable used along a path without calls to a callee-save register when a caller-save register could be used instead. Our approach uses inevitable-call regions to guide the placement of variables into registers and may decide to keep a variable in a caller-save register until a call is inevitable, at which point it may be moved into a callee-save register.

3 Implementation

We allocate n registers for use by our register allocator. Two of these are used for the return address and closure pointer. For some fixed $c \leq n-2$, the first c actual parameters of all procedure calls are passed via these registers; the remaining parameters are passed on the stack. When evaluating the arguments to a call, unused registers are used when "shuffling" is necessary because of cycles in the dependency graph. We also fix a number $l \leq n-2$ of these registers to be used for user variables and compiler-generated temporaries.

The lazy save placement algorithm requires two linear passes through the abstract syntax tree. The first pass performs greedy shuffling and live analysis, computes $\mathcal{S}_t[E]$ and $\mathcal{S}_f[E]$ for each expression, and introduces saves. The second pass removes redundant saves.

The eager restore placement algorithm requires one linear pass. It computes the "possibly referenced before the next call" sets and places the restores. This pass can be run in parallel with the second pass of the lazy save placement algorithm, so the entire register allocation process requires just two linear passes.

3.1 Pass 1: Save Placement

The first pass processes the tree bottom-up to compute the live sets and the register saves at the same time. It takes two inputs: the abstract syntax tree (T) and the set of registers live on exit from T. It returns the abstract syntax tree annotated with register saves, the set of registers live on entry to T, $\mathcal{S}_t[T]$, and $\mathcal{S}_f[T]$.

[2] This was in error in the proceedings.

Liveness information is collected using a bit vector for the registers, implemented as an n-bit integer. Thus, the union operation is logical or, the intersection operation is logical and, and creating the singleton $\{r\}$ is a logical shift left of 1 for r bits.

Save expressions are introduced around procedure bodies and the "then" and "else" parts of **if** expressions, unless both branches require the same register saves.

When we encounter a call, we use the following greedy shuffling algorithm:

1. We build the dependency graph for the register arguments. Since calls destroy the argument registers, building the dependency graph involves traversing the tree only down to calls. After the order of evaluation has been determined, the arguments are traversed again by the current pass, which will visit the nodes only one more time, since they require no shuffling. Thus, the overall pass is linear since all nodes in the tree are visited at most twice.

2. We partition the register arguments into those that do not contain calls (simple) and those that do (complex).

3. We search through the complex arguments for one on which none of the simple arguments depend. If one is found, it is placed onto the simple list. If not, the last complex argument is placed onto the simple list. The remaining complex arguments are evaluated and placed into temporary stack locations, since evaluation of complex arguments may require a call, causing the previous arguments to be saved on the stack anyway.

4. We look for an argument in the simple list that has no dependencies on the remaining argument registers. If we find one, we push it onto a "to be done last" stack of arguments. Then we remove it from the dependency graph and simple list and repeat step 4 until all arguments have been assigned. Once all have been assigned, we evaluate the arguments in the "to be done last" stack directly into their argument registers. We conclude by assigning all the remaining argument registers from the corresponding temporary locations.

5. If all arguments have dependencies on each other, we greedily pick the one that causes the most dependencies and evaluate it into a temporary location, in hopes of breaking all the cycles. After removing it from the dependency graph and simple list, we continue with step 4. We use other argument registers as temporaries when possible; otherwise, we use the stack.

This algorithm is $O(n^3)$, where n is the number of argument registers. Fortunately, n is fixed and is usually very small, *e.g.*, six in our current implementation. Consequently, it does not affect the linearity of the pass. Furthermore, n is limited in practice by the number of simple register arguments, and the operations performed involve only fast integer arithmetic. Finding the ordering of arguments that minimizes the number of temporaries is NP-complete. We tried an exhaustive search and found that our greedy approach works optimally for the vast majority of all cases, mainly because most dependency graph cycles are simple. In our benchmarks, only 7% of the call sites had cycles. Furthermore, the greedy algorithm was optimal for all of the call sites in all of the benchmarks excluding our compiler, where it was optimal in all but six of the 20,245 call sites, and in these six it required only one extra temporary location.

3.2 Pass 2: Save Elimination and Restore Placement

The second pass processes the tree to eliminate redundant saves and insert the restores. It is significantly simpler than the first pass. It takes three inputs: the abstract syntax tree (T), the current save set, and the set of registers possibly referenced after T but before the next call. It returns two outputs: the abstract syntax tree with redundant saves eliminated and restores added, and the set of registers possibly referenced before the next call.

When a save that is already in the save set is encountered, it is eliminated. Restores for possibly referenced registers are inserted immediately after calls.

Our earlier example demonstrates why there may be redundant saves. Let's assume that we have a procedure body of (**seq** (**if** (**if** x **call** false) y **call**) x). Then the first pass of the algorithm would introduce saves as follows:

```
(save (x)
      (seq (if (if x (save (x y) call) false)
               y
               (save (x) call))
           x))
```

Two of the saves are redundant and can be eliminated. The result of the second pass would be:

```
(save (x)
      (seq (if (if x
                   (save (y)
                         (restore-after call (x y)))
                   false)
               y
               (restore-after call (x)))
           x))
```

4 Performance

To assess the effectiveness of our register allocator we measured its ability to eliminate stack references and its impact on execution time for the set of programs described in Table 1 and for the Gabriel benchmarks. We also examined the effect of our lazy save placement versus the two natural extremes, "early" and "late." Although the early strategy eliminates all redundant saves, it generates unnecessary saves in non-syntactic leaf routines. Because the late save strategy places register saves immediately before calls, it handles all effective leaf routines well. This late strategy, however, generates redundant saves along paths with multiple calls. Our lazy strategy strikes a balance between the two extremes by saving as soon as a call is inevitable. Consequently, it avoids saves for all effective leaf routines and at the same time eliminates most redundant saves.

For the baseline and comparison cases local register allocation was performed by the code generator, eight registers were globally allocated to support our run-time model (which provides in-line allocation, fast access to free variables, *etc.*), and our greedy shuffling algorithm was employed. Thus our baseline for comparison is a compiler that already makes extensive use of registers. We were able to collect data on stack references by modifying our compiler to instrument programs with code to count stack accesses.

Table 3 shows the reduction in stack references and CPU time when our allocator is permitted to use six argument registers. The table also gives the corresponding figures for the early and late save strategies. On average, the lazy save approach eliminates 72% of stack accesses and increases

	Lazy Save		Early Save		Late Save	
Benchmark	stack ref reduction	performace increase	stack ref reduction	performace increase	stack ref reduction	performace increase
Compiler	72%	30%	64%	26%	63%	13%
DDD	68%	47%	59%	43%	63%	44%
Similix	69%	26%	57%	18%	51%	9%
SoftScheme	71%	22%	55%	14%	43%	5%
boyer	66%	54%	49%	46%	60%	21%
browse	72%	39%	60%	35%	70%	56%
cpstak	77%	35%	59%	26%	77%	20%
ctak	71%	85%	50%	20%	70%	64%
dderiv	56%	52%	46%	42%	44%	47%
destruct	88%	33%	82%	34%	87%	36%
div-iter	100%	133%	80%	99%	99%	130%
div-rec	77%	30%	65%	29%	76%	38%
fft	71%	19%	71%	2%	67%	–3%
fprint	68%	15%	48%	9%	61%	16%
fread	67%	39%	56%	22%	58%	24%
fxtak	62%	35%	32%	14%	45%	24%
fxtriang	76%	43%	59%	47%	76%	15%
puzzle	74%	34%	68%	32%	74%	34%
tak	72%	109%	52%	92%	50%	93%
takl	86%	67%	58%	41%	83%	67%
takr	72%	15%	52%	3%	50%	4%
tprint	63%	11%	41%	8%	56%	11%
traverse-init	76%	38%	70%	37%	74%	44%
traverse	50%	29%	48%	28%	48%	30%
triang	83%	41%	71%	32%	75%	48%
Average	72%	43%	58%	32%	65%	36%

Table 3. Reduction of stack references and resulting speedup for three different save strategies given six argument registers relative to the baseline of no argument registers. For both baseline and comparison cases local register allocation was performed by the code generator, several registers were globally allocated to support the run-time model, and our greedy shuffling algorithm was employed.

	`cc -O3`	`gcc -O3`	*Chez Scheme*
Speedup	0%	5%	14%

Table 4. Performance comparison of *Chez Scheme* against the GNU and Alpha OSF/1 optimizing C compilers for `tak(26, 18, 9)` benchmark (results are normalized to the Alpha OSF/1 compiler).

		Early Save	Lazy Save	Speedup
Callee save	`cc -O3`	1.292s	0.676s	91%
	`gcc -O3`	1.233s	0.772s	60%
Caller-save	by hand	0.990s	0.638s	55%

Table 5. Execution times of optimized C code for `tak(26, 18, 9)` using early and lazy save strategies for callee-save registers, and hand-coded assembly using lazy saves for caller-save registers.

run-time performance by 43%, a significant improvement over both the early (58%/32%) and late (65%/36%) save approaches.

To compare our strategy against other register allocators, we timed Scheme-generated code for the call-intensive `tak` benchmark against fully optimized code generated by the GNU and Alpha OSF/1 C compilers. Table 4 summarizes these results using the Alpha C compiler as a baseline. We chose the `tak` benchmark because it is short and isolates the effect of register save/restore strategies for calls and because the transliteration between the Scheme and C source code is immediately apparent. Our Scheme code actually outperforms optimized C code for this benchmark despite the additional overhead of our stack overflow checks [11] and poorer low-level instruction scheduling.

Part of the performance advantage for the Scheme version is due to our compiler's use of caller-save registers, which turns out to be slightly better for this benchmark. The remaining difference is due to the lazy save strategy used by the Scheme compiler versus the early saves used by both C compilers.

To study the effectiveness of our save strategy for both caller- and callee-save registers, we hand-modified the optimized assembly output of both C compilers to use our lazy save technique. In order to provide another point of comparison, we hand-coded an assembly version that uses caller-save registers. Table 5 compares the original C compiler times with the modified versions and the hand-coded version. In all cases the lazy save strategy is clearly beneficial and brings the performance of the callee-save C code within range of the caller-save code.

To measure the effect of the additional register allocation passes on compile time, we examined profiles of the compiler processing each of the benchmark programs and found that register allocation accounts for an average of 7% of overall compile time. This is a modest percentage of run time for a compiler whose overall speed is very good: on an Alpha 3000/600, *Chez Scheme* compiles itself (30,000 lines) in under 18 seconds. In contrast, the Alpha OSF/1 C compiler requires 23 seconds to compile the 8,500 lines of support code used by our implementation. While 7% is acceptable overhead for register allocation, the compiler actually runs faster with the additional passes since it is self-compiled and benefits from its own register allocator.

We also ran the benchmarks with several other variations in the numbers of parameters and user variables permitted to occupy registers, up through six apiece. Performance increases monotonically from zero through six registers, although the difference between five and six registers is minimal. Our greedy shuffling algorithm becomes important as the number of argument registers increases. Before we installed this algorithm, the performance actually decreased after two argument registers.

5 Related Work

Graph coloring [5] has become the basis for most modern register allocation strategies. Several improvements to graph coloring have been made to reduce expense, to determine which variables should receive highest priority for available registers, and to handle interprocedural register allocation.

Steenkiste and Hennessy [14] implemented a combined intraprocedural and interprocedural register allocator for Lisp that assigns registers based on a bottom-up coloring of a simplified interprocedural control flow graph. They handle cycles in the call graph and links to anonymous procedures by introducing additional saves and restores at procedure call boundaries. Using a combination of intraprocedural and interprocedural register allocation, they are able to eliminate 88% of stack accesses; approximately 51% via intraprocedural register allocation and the remainder via interprocedural allocation. Our figure of 72% appears to compare favorably since we do not perform any interprocedural analysis; differences in language, benchmarks, and architecture, however, make direct comparison impossible.

Steenkiste and Hennessy found that an average of 36% of calls at run time are to (syntactic) leaf routines; this is similar to our findings of an average slightly below one third. They did not identify or measure the frequency of calls to effective leaf routines.

Chow and Hennessy [7] present an intraprocedural algorithm that addresses certain shortcomings of straightforward graph coloring. In their approach, coloring of the register interference graph is ordered by an estimate of total run-time savings from allocating a live range to a register, normalized by the size of the region occupied. For live ranges with equal total savings, priority goes to the shorter in hopes that several short live ranges can be allocated to the same register. In order to improve procedure call behavior, incoming and outgoing parameters are "pre-colored" with argument registers. The priority-coloring algorithm is able to make effective use of caller-save registers for syntactic leaf procedures, preferring callee-save registers for the rest.

Chow [6] extends the priority-based coloring algorithm to an interprocedural register allocator designed to minimize register use penalties at procedure calls. He provides a mechanism for propagating saves and restores of callee-save registers to the upper regions of the call graph. In this scheme, saves and restores propagate up the call graph until they reach a procedure for which incomplete information is available due to cycles in the call graph, calls through function pointers, or procedures from external modules. Such restrictions render this approach ineffective for typical Scheme programs which rely on recursion and may make extensive use of first-class anonymous procedures. Chow also claims that interprocedural register allocation requires a large number of registers in order to have a noticeable impact; the 20 available to his algorithm were inadequate for large benchmarks.

Clinger and Hansen [9] describe an optimizing compiler for Scheme which achieves respectable performance through a combination of aggressive lambda-lifting and parallel assignment optimization. Lambda lifting transforms the free variables of a procedure into extra arguments, decreasing closure creation cost and increasing the number of arguments subject to register allocation. Parallel assignment optimization then attempts to make passing arguments in registers as efficient as possible by selecting an order of evaluation that minimizes register shuffling. Their shuffling algorithm is similar to ours in that it attempts to find an ordering that will not require the introduction of temporaries but differs in that any cycle causes a complete spill of all arguments into temporary stack locations. Although [12] describes a register shuffling algorithm similar to ours, details regarding the selection of the node used to break cycles are not given.

Shao and Appel [13, 1] have developed a closure conversion algorithm that exploits control and data flow information to obtain extensive closure sharing. This sharing enhances the benefit they obtain from allocating closures in registers. Graph-coloring global register allocation with careful lifetime analysis allows them to make flexible and effective use of callee- and caller-save registers. Since the order of argument evaluation is fixed early in their continuation-passing style compiler, they attempt to eliminate argument register shuffling with several complex heuristics including choosing different registers for direct-called functions.

6 Conclusions and Future Work

In this paper we have described a fast linear intraprocedural register allocation mechanism based on lazy saves, eager restores, and greedy shuffling that optimizes register usage across procedure calls. The lazy save technique generates register saves only when procedure calls are inevitable, while attempting to minimize duplicate saves by saving as soon as it can prove that a call is inevitable. We have shown that this approach is advantageous because of the high percentage of effective leaf routines. The eager restore mechanism restores immediately after a call all registers possibly referenced before the next call. While a lazy restore mechanism would reduce the number of unnecessary restores, we found that restoring as early as possible compensates for unnecessary restores by reducing the effect of memory latency. The greedy shuffling algorithm orders arguments and attempts to break dependency cycles by selecting the argument causing the most dependencies; it has proven remarkably close to optimal in eliminating the need for register shuffling at run time.

Our performance results demonstrate that around 72% of stack accesses are eliminated via this mechanism, and run-time performance increases by around 43% when six registers are available for parameters and local variables. Our baseline for comparison is efficient code generated by an optimizing compiler that already makes extensive use of global registers and local register allocation. This is within range of improvements reported for interprocedural register allocation [14, 6]. Although the compiler now spends around 7% of its time on register allocation, the compiler actually runs faster since it is self-compiled and benefits from its own register allocation strategy.

Our effective leaf routine statistics suggest a simple strategy for static branch prediction in which paths without calls are assumed to be more likely than paths with calls. Preliminary experiments suggest that this results in a small (2–3%) but consistent improvement.

Other researchers have investigated the use of lambda lifting to increase the number of arguments available for placement in registers [13, 9]. While lambda lifting can easily result in net performance decreases, it is worth investigating whether lambda lifting with an appropriate set of heuristics such as those described in [13] can indeed increase the effectiveness of our register allocator without significantly increasing compile time.

Acknowledgement: The authors would like to thank Mike Ashley for his helpful comments on an earlier version of this paper.

References

[1] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5(3):191–221, 1992.

[2] Anders Bondorf. *Similix Manual, System Version 5.0.* DIKU, University of Copenhagen, Denmark, 1993.

[3] Bhaskar Bose. DDD—A transformation system for Digital Design Derivation. Technical Report 331, Indiana University, Computer Science Department, May 1991.

[4] Robert G. Burger. The Scheme Machine. Technical Report 413, Indiana University, Computer Science Department, August 1994.

[5] G. J. Chaitin, M. A. Auslander, A. K. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[6] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[7] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[8] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[9] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, 1994.

[10] Richard P. Gabriel. *Performance and Evaluation of LISP Systems.* MIT Press series in computer systems. MIT Press, Cambridge, MA, 1985.

[11] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[12] David Kranz. Orbit: an optimizing compiler for Scheme. Technical Report 632, Yale University, Computer Science Department, 1988.

[13] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 150–161, 1994.

[14] P. A. Steenkiste and J. L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for Lisp. *Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[15] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, 1994.