# EFFICIENT COMPILATION AND PROFILE-DRIVEN DYNAMIC RECOMPILATION IN SCHEME

Robert G. Burger

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

March 1997

# Acknowledgments

*You are worthy, our Lord and God,*
*to receive glory and honor and power,*
*for you created all things,*
*and by your will they were created*
*and have their being.*

*Revelation 4:11*

First and foremost I thank Jesus, my King and Redeemer, who graciously enabled me to complete this dissertation by providing the support acknowledged below.

I am very grateful to Kent Dybvig, my outstanding research adviser, for his good ideas, advice, and encouragement. I also appreciate the excellent teaching and support that Dan Friedman, Frank Prosser, and Larry Moss have given me.

I thank the National Science Foundation for supporting me for three years on a graduate fellowship and for a year as a research assistant. I also appreciate the Computer Science Department's support, especially the invaluable experience as associate instructor for the Scheme compiler classes.

I appreciate Oscar Waddell's help, especially his contributions to register allocation, source-object correlation, and graphical display of data.

I am grateful to Claude Anderson for all his encouragement, especially for convincing me that I really could and should complete this dissertation!

I thank my parents, Ron and Dianne Burger, and especially my wife, Stacy, for their unwavering support and love, for listening to me, for encouraging me, and for helping me keep things in perspective.

# Abstract

This dissertation presents a fast and effective linear intraprocedural register allocation strategy and an infrastructure for profile-driven dynamic recompilation in Scheme. The register allocation strategy 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.

The infrastructure for profile-driven dynamic recompilation enables the run-time system to recompile procedures—even while they are executing—using dynamically collected profile data. In current programming environments, profile-based recompilation requires a tedious compile-profile-recompile cycle. In our system, the instrumentation for profiling and the subsequent recompilation are done at run time. As a proof of concept, edge-count profile data is used to reorder basic blocks to reduce the number of mispredicted branches and instruction cache misses. Our low-overhead profiling strategy supports first-class continuations and reinstrumentation of active procedures. It includes a fast and effective linear static edge-count estimator that accurately predicts common run-time checks. In addition, this dissertation demonstrates how the profile data can be graphically associated with the original source to provide useful feedback to programmers.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This dissertation presents a fast and effective linear intraprocedural register allocation strategy and an infrastructure for profile-driven dynamic recompilation in Scheme. The register allocation strategy optimizes register usage across procedure calls. This area of optimization is particularly important for languages in which procedure calls rather than explicit looping constructs are used as the basis for control.

The infrastructure for dynamic recompilation enables the run-time system to recompile procedures, even while they are executing. In order for recompilation to produce code improvements, some dynamic information unavailable at compile time must be employed. As a proof of concept, this dissertation uses edge-count profile data to provide the basis for dynamic recompilation that optimizes profile counter placement and minimizes the number of mispredicted branches and instruction cache misses by reordering basic blocks.

The low-overhead edge-count profile strategy supports first-class continuations and reinstrumentation of active procedures. It employs a fast log-linear algorithm that determines optimal counter placement and a fast linear static edge-count estimator that significantly improves initial counter placement. In addition, this dissertation describes a graphical viewer that allows programmers to view the profile data in terms of the original source.

The remainder of this chapter is organized as follows. The next section gives an overview of our register allocation strategy. Section 1.2 gives an overview of edge-count profiling and graphical display of the data. Section 1.3 gives an overview of the infrastructure for dynamic recompilation using basic block reordering as a proof of concept.

The remainder of this dissertation is organized similarly. Chapter 2 describes our register allocation strategy. Chapter 3 presents our low-overhead edge-count profiling strategy and describes how the profile data can be graphically associated with the original source. Chapter 4 demonstrates the feasibility and utility of dynamic recompilation by applying profile data to reorder basic blocks and optimize profile counter placement. The final chapter summarizes the contributions of this dissertation and lists areas for future work.

## 1.1 Register Allocation

Chapter 2 presents our fast and effective linear intraprocedural register allocation strategy that optimizes register usage across procedure calls. It capitalizes on the 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.

Traditional save placement involves two extremes. On the one hand, the callee-save registers used in a procedure are saved on entry to the procedure. On the other hand, the caller-save registers containing information needed after a call are saved just before the call. The former strategy saves "too early" in the sense that some registers may be saved even though there is a path through the procedure that does not use them. The latter strategy saves "too late" in the sense that it may introduce redundant saves when a path contains multiple calls.

Our strategy avoids both extremes by saving registers only when a call is inevitable (not too early) and as soon as a call is inevitable (not too late). This lazy save strategy benefits the large class (approximately one third) of procedure activations that make no calls even though their procedure bodies contain calls, and it does not unduly penalize procedure activations that make multiple calls.

The save placement algorithm is based on a recursive definition of the set of registers to save around an expression. The algorithm avoids placing saves too early by taking the intersection of the save sets for the "then" and "else" branches of **if** statements. It avoids placing saves too late by propagating the save sets up the control-flow graph using the union operator.

Because Scheme uses **if** expressions to express short-circuit boolean operations such as **and** and **or**, the save placement algorithm must be sensitive to conditionals and constants, especially when they occur in a conditional context. Consider the expression (**if** (**and** $x$ $(f)$) $y$ $(g)$), which expands to (**if** (**if** $x$ $(f)$ #f) $y$ $(g)$). Every path through this expression makes a procedure call. A simple lazy save strategy, however, would be fooled into thinking that the call-free path from $x$ to #f to $y$ can occur. Consequently, it will not propagate the saves as desired.

To correct this deficiency, the save placement algorithm divides the register save sets into the two classes used by **if** statements: true and false. The distinction allows the algorithm to have a more precise notion of the paths through the control-flow graph. Along a given path it takes the union of the registers that need to be saved. Then it takes the intersection of all the paths to determine the registers that need to be saved regardless of the path taken. For example, if #f is used in a conditional context, its true save set is defined to be the identity under intersection so that the impossible path down the "then" branch does not unnecessarily restrict the save set for the entire **if**.

Using the same "as soon as but no sooner than necessary" principle, we tried a lazy restore placement strategy. We compared the performance results against the simpler, straightforward eager restore placement strategy of immediately restoring any register possibly referenced before the next call. Even though the eager strategy results in unnecessary restores, it produced code that ran just as fast as the code produced by the lazy strategy. We concluded that the reduced effect of memory latency offsets the cost of unnecessary restores, and latency continues to get worse as processors become faster and memory hierarchies become deeper. We therefore use the eager restore strategy.

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, the call to $f$ in (**lambda** $(x\ y)$ $(f\ y\ x)$) requires a swap of the registers for $x$ and $y$. 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.

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 [20, 29]. 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, but we have observed that it finds the optimal ordering for a vast majority of call sites.

## 1.2 Edge-Count Profiling

Chapter 3 extends the optimal edge-count profiling strategy described by Ball and Larus [4] to support first-class continuations and reinstrumentation of active procedures. Our strategy employs a fast log-linear algorithm to determine optimal counter placement, and it significantly improves initial counter placement using a fast linear static edge-count estimator. The profile data is used for two purposes. The primary application is for the dynamic recompiler, which uses the data to perform run-time optimizations such as basic block reordering and to minimize the overhead of profiling by optimizing counter placement. The secondary application is for the programmer, who can display that data with a graphical annotation of the original source.

Edge-count profiling is based on the concept of a basic block, a sequential list of instructions for which control enters only at the top and exits only at the bottom. A procedure is composed of one or more basic blocks, depending on its control structure. The branches in a procedure determine how the blocks are connected. Consequently, the entire procedure can be viewed as a directed graph whose nodes are the basic blocks and whose edges come from the branches.

The goal of edge-count profiling is to measure the number of times each edge in the control-flow graph is executed. Because this information is directly related to the branches, it is ideally suited for branch prediction and block reordering. In addition, the number of times each basic block is executed, which can be readily computed from the edge counts, can be associated with the original source to give the programmer feedback.

Ball and Larus augment the control-flow graph with an explicit exit node to account for all possible control-flow paths [4]. As a result, the control-flow graph satisfies the conservation of flow property. This key property allows them to use a maximal spanning tree to avoid placing counters on the most frequently executed edges. Instead, the counts for these edges are computed after a program run from the counts of the instrumented edges.

The conservation of flow property is met only under the assumption that each procedure call returns exactly once. First-class continuations, however, cause some procedure activations to exit prematurely and others to be reinstated one or more times. Even when there are no continuations, reinstrumenting a procedure while it has activations on the stack is problematic because some of its calls have not returned yet.

Ball and Larus handle the restricted class of exit-only continuations by adding to each call block an edge pointing to the exit block [4]. They measure the counts of exit edges directly by modifying continuation invocation to account for aborted activations. Their strategy can be extended to support fully general continuations by modifying continuation invocation to account for reinstated activations as well as aborted ones. We use a similar strategy, but to avoid the linear stack walk that would destroy constant-time continuation invocation, we measure the counts indirectly by ensuring that exit edges are on the spanning tree.

When a procedure is compiled for the first time, all its edge counts are zero, so the maximal spanning tree algorithm finds a spanning tree that may be far from maximal when the procedure actually executes. Our static estimator significantly improves initial counter placement by simulating profile data. It correctly predicts the behavior of common run-time tests such as heap and stack overflow, identifies and predicts internal loops, and assumes all other tests have a 50% chance of succeeding. Unlike Ball and Larus's estimator [4], it is linear in the number of basic blocks and does not require the control-flow graph to be reducible. It capitalizes on the observation that the counts of the control-flow graph need not satisfy the conservation of flow property to determine counter placement.

The estimator performs a depth-first traversal of the control-flow graph to identify and mark the back edges (internal loops) and to determine a block ordering used to propagate the weights. Using this block ordering, the algorithm then processes each block, computing the total of the incoming flow and dividing it among the outgoing edges. When the algorithm recognizes a block that performs a common run-time test, it divides the flow to favor the outgoing edge that is more likely to be executed. Otherwise, it divides the flow evenly. The conservation of flow property is violated only when there are back edges.

Even though edge counts are at just the right level for the recompiler, they are at too low a level for the programmer, who would prefer to see them in terms of the original source. Consequently, we modified our system's macro expander to collect source information and propagate it through macro expansion. This information is further propagated through the various passes of the compiler until it finally appears as special assembly-language pseudo-instructions. These pseudo-instructions are stored in the basic block structures so that the block counts can be associated with the original source. The Scheme Widget Library [46] provides the tools necessary to display the source with graphical annotations of the counts.

## 1.3 Dynamic Recompilation

Chapter 4 describes an infrastructure for dynamic recompilation in Scheme using the example of basic block reordering based on edge-count profile data. Because in Scheme the garbage collector can find and relocate all references to any given object, it is ideally suited for completely and transparently replacing a procedure with a new-and-improved version of it. As a result, procedures can be recompiled at run time, even when they have activations on the stack or in captured continuations.

To demonstrate the feasibility and utility of dynamic recompilation, we employ a well-established optimization, basic block reordering [14, 35, 36]. Current computer architectures increase performance by predicting and prefetching instructions from a large, fast cache. As a result, the penalty for a mispredicted branch or an instruction cache miss can be quite large. Reordering basic blocks to minimize the number of mispredicted branches and improve code locality has thus become a significant way for compilers to improve performance. Although there are various heuristic algorithms that estimate branch behavior statically, profile-based prediction is significantly better at reducing the number of mispredicted branches [5, 47].

We use a variant of Pettis and Hansen's basic block reordering algorithm [35]. The reordering proceeds in two steps. First, blocks are combined into chains according to the most frequently executed edges to reduce the number of instruction cache misses. Second, the chains are ordered based on the target architecture's branch prediction strategy to reduce the number of mispredicted branches.

In existing programming environments, basic block reordering has been done at compile time and requires a separate run to collect profile data from sample inputs. We demonstrate that basic block reordering and the insertion and removal of profile instrumentation can be done transparently at run time, eliminating the tedious compile-profile-recompile cycle. Programmers have complete control over which procedures to instrument and when to recompile them, but they may choose to have the system make these decisions for them.

Our infrastructure enables user-definable automatic recompilation. For example, one can specify that at every garbage collection, all profiled procedures executed more than 1000 times will be recompiled with instrumentation off. As a result, programs run faster and faster as the procedures are automatically recompiled. The resulting heap with the recompiled procedures can be saved for production use.

In order to support dynamic recompilation, we associate with each procedure a symbolic representation of its basic blocks and connecting edges (branches), including edge counts and source information. This structure allows the compiler to generate and regenerate the machine code for the procedure as well as to provide count information to the source viewer.

When procedures are recompiled, new procedures are created and linked to the original ones. During the next garbage collection, the original procedures are replaced by the new ones. Using a translation table associated with each recompiled procedure, the collector properly relocates all entry and return points whose offsets may have changed during recompilation. Because the collector translates return addresses, procedures can be recompiled while they are executing. In addition, the collector updates the counts of uninstrumented edges as needed to account for changes in counter placement so that the counts remain accurate across recompilation.

In environments where it is generally impossible to locate all pointers to a given procedure at run time, dynamic recompilation can be implemented by introducing one level of indirection into procedure calls and returns. For each procedure call, the compiler can generate a jump to an entry stub that in turn jumps to the actual entry point. Similarly, the compiler can cause each procedure call to return through a return stub that jumps to the appropriate return point. With this design, translating entry and return points is as simple as changing the corresponding stubs. The extra branch may not reduce performance significantly, especially in systems with a low frequency of procedure calls or with a similar mechanism already in place for dynamic linking. With branch-folding hardware, there would be no overhead, because the branches would be eliminated from the pipeline.

Dynamic recompilation need not be limited to low-level optimizations such as block reordering. By associating profile data with earlier compilation passes, the recompiler can perform run-time optimizations involving register allocation, flow anaylsis, lambda lifting, and procedure inlining. Because these optimizations destroy the one-to-one correspondence among basic blocks, the collector must be sensitive to return points in original procedures that have no corresponding, compatible return points in recompiled procedures. By retaining original procedures as long as they have live unassociated return addresses, the collector can support these optimizations—even on running procedures.

# Chapter 2

# Register Allocation

## 2.1 Background and Overview

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 [13]. 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 chapter 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.

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.

The remainder of this chapter is organized as follows. Section 2.2 presents our lazy save, eager restore, and greedy shuffling algorithms. Section 2.3 describes our implementation of the algorithms. Section 2.4 discusses the performance characteristics of our implementation. Section 2.5 describes related work.

## 2.2 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 [19] below. In Section 2.2.4 we explain how our strategy applies to callee-save registers.

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

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**.

### 2.2.1 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 register saves between these two extremes for both callee- and caller-save registers.

| *Benchmark* | *Lines* | *Description* |
|---|---|---|
| compiler | 30,778 | *Chez Scheme* 5.0b recompiling itself |
| ddd | 9,578 | Digital Design Derivation System 1.0 [7] deriving hardware for a Scheme machine [9] |
| similix | 7,305 | self-application of the Similix 5.0 partial evaluator [6] |
| softscheme | 10,073 | Andrew Wright's soft typer [50] checking his pattern matcher |

Table 1: Description of register allocation benchmarks

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 [26]. 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. We then present an improved algorithm that handles these cases.

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

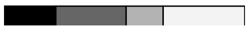
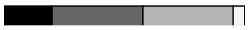
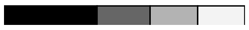
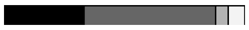
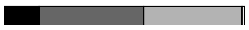
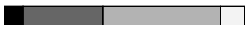
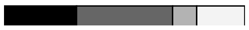
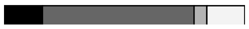
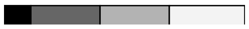
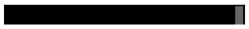
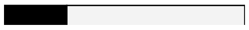
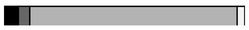
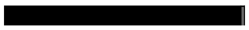
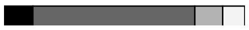
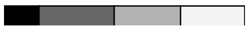
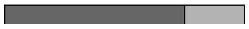
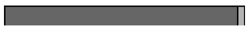
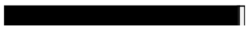
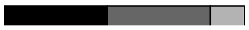
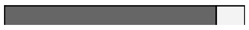
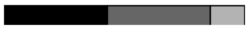
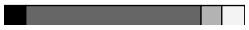
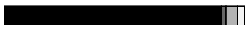
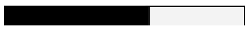
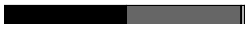
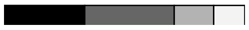
| *Benchmark* | *Calls* | *Breakdown* |
|---|---|---|
| compiler | 33,041,034 |  |
| ddd | 86,970,102 |  |
| similix | 33,891,834 |  |
| softscheme | 11,153,705 |  |
| boyer | 914,113 |  |
| browse | 1,608,975 |  |
| cpstak | 159,135 |  |
| ctak | 302,621 |  |
| dderiv | 191,219 |  |
| destruct | 236,412 |  |
| 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

| | |
|---|---|
| ■ | syntactic leaf nodes |
| ▓ | non-syntactic leaf nodes |
| ▒ | non-syntactic internal nodes |
| □ | syntactic internal nodes |

### A Simple Save Placement Algorithm

We 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 $\mathcal{S}[E] = \emptyset$.

### 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:

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

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

## The Revised Save Placement Algorithm

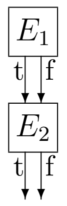
The basic principle is to consider the paths through the control flow graph. Along a given path we take the union of the registers that need to be saved at each node. Then we take the intersection of all the paths to determine the registers that need to be saved regardless of the path taken. In order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

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

Since it is impossible that $\mathsf{true}$ should evaluate to $\mathsf{false}$, 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.

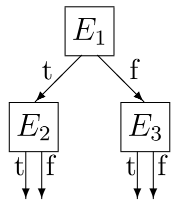
Now we define the recursive cases. Intuitively, the set $\mathcal{S}_t[(\textbf{seq}\ E_1\ E_2)]$ is the set of registers to save if the **seq** expression evaluates to $\mathsf{true}$. There are two possible paths: $E_1$ is $\mathsf{true}$ and $E_2$ is $\mathsf{true}$, or $E_1$ is $\mathsf{false}$ and $E_2$ is $\mathsf{true}$. 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:



$(\textbf{seq}\ E_1\ E_2)$

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

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



$(\textbf{if}\ E_1\ E_2\ E_3)$

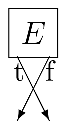
$$\mathcal{S}_t[\textbf{if}] = (\mathcal{S}_t[E_1] \cup \mathcal{S}_t[E_2]) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_t[E_3])$$
$$\mathcal{S}_f[\textbf{if}] = (\mathcal{S}_t[E_1] \cup \mathcal{S}_f[E_2]) \cap (\mathcal{S}_f[E_1] \cup \mathcal{S}_f[E_3])$$

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

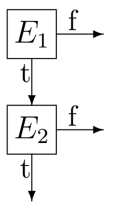
$$\begin{aligned} \mathcal{S}_t[B] &= (\emptyset \cup (\{y\} \cup L)) \cap (\emptyset \cup R) & \mathcal{S}_f[B] &= (\emptyset \cup (\{y\} \cup L)) \cap (\emptyset \cup \emptyset) \\ &= \{y\} \cup L & &= \emptyset \end{aligned}$$

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

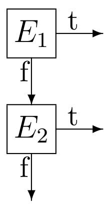
We see that although no registers would be saved around the inner **if** expression (since $\mathcal{S}_t[B] \cap \mathcal{S}_f[B] = \emptyset$), all the live registers would be saved around the outer **if** as desired.



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



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



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

Figure 1: Save placement graphs and equations for boolean operators

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.

### 2.2.2 Eager Restore Placement

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

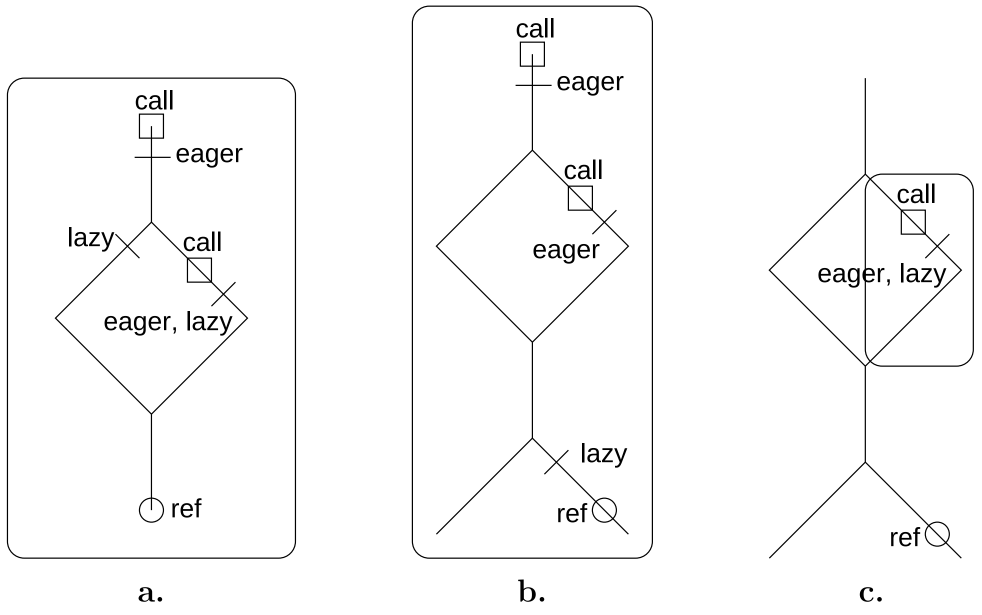


Figure 2: Abbreviated control-flow graphs for three restore scenarios

- 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.

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.

### 2.2.3 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 [20, 29]. 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 2.3.1).

### 2.2.4 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 holds the return address of the current procedure and must be saved and restored around calls. Consequently, if $ret \in \mathcal{S}_t[E] \cap \mathcal{S}_f[E]$, 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 [17] 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.

## 2.3 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, both of which appear as **let** expressions at this point in the compiler.

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.

### 2.3.1 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 $\mathcal{S}_f[T]$.

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.

We recognize **let** expressions and handle them separately. If some of the $l$ registers are not live on entry to the body of the **let**, they are allocated in a first-come, first-served basis to the **let** variables excluding those bound to **lambda** expressions. Because our compiler implements direct calls to **let**-bound procedures, allocating registers to them would be fruitless. At this point in the pass, the compiler has not yet determined the set of registers live on entry to the body of the **let**. In an earlier pass, however, it could have already collected the set of free variables of the **let** expression. Thus, it can take the union of this set and the set of registers live on exit to determine the set of registers live on entry.

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. In two of these cases a register was available for this extra location.

### 2.3.2 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. Suppose 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 ($\underline{x}$ y) call) false)
           y
           (save ($\underline{x}$) call))
       x))
```

Two of the saves are redundant (and are underlined) 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))
```

## 2.4 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 Scheme versions of 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 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 assembly code generated by Scheme 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

| | *Lazy Save* | | *Early Save* | | *Late Save* | |
|---|---|---|---|---|---|---|
| *Benchmark* | *stack ref reduction* | *perf. increase* | *stack ref reduction* | *perf. increase* | *stack ref reduction* | *perf. 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 performance increase 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* |
|---|---|---|---|
| *Speed-up* | 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)`

| | | *Early Save* | *Lazy Save* | *Speed-up* |
|---|---|---|---|---|
| *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

apparent. Our Scheme code actually outperforms optimized C code for this benchmark despite the additional overhead of our stack overflow checks [28] 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 save strategy 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* 5.0b compiles itself (30,778 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 [23].

## 2.5 Related Work

Graph coloring [13] has become the basis for most register allocation strategies today. 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 [44] 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 [18] 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 [17] 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 [20] 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 Kranz [29] 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 [39, 2] 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 heuristics including choosing different registers for known functions.

# Chapter 3

# Edge-Count Profiling

This chapter describes a low-overhead edge-count profiling strategy that supports first-class continuations and reinstrumentation of active procedures. It is based on one described by Ball and Larus [4] that minimizes the total number of profile counter increments done at run time. It extends Ball and Larus's strategy with support for first-class continuations and reinstrumentation of active procedures. In addition, our strategy provides a fast linear static edge-count estimator that significantly improves initial counter placement, and it employs a fast log-linear algorithm to determine optimal counter placement.

There are two main uses for profile data. First, the dynamic recompiler (see Chapter 4) uses it to perform run-time optimizations such as basic block reordering and to optimize counter placement. Second, it is used to give feedback to the programmer via graphical annotation of the original source code.

Section 3.1 reviews Ball and Larus's optimal edge-count placement algorithm [4]. Section 3.2 presents modifications to support first-class continuations and reinstrumentation of active procedures. Section 3.3 describes our implementation. Section 3.4 presents our linear static edge-count estimator. Section 3.5 explains how the profile data is correlated with the original source. Section 3.6 reports the run-time and compile-time costs of profiling. Section 3.7 describes related work.

## 3.1 Background

A given procedure is represented by a control-flow graph composed of basic blocks and weighted edges. The assembly language instructions for the procedure are split into basic blocks, which are sequential sections of code for which control enters only at the top and exits only from the bottom. The branches at the bottoms of the basic blocks determine how the blocks are connected, so they become the edges in the graph. If there are blocks that cannot be reached by any path from the entry points of the procedure, they are removed from the control-flow graph so that the resulting graph is connected. The weight of each edge represents the number of times the corresponding branch is taken.

The key property needed for optimal profiling is conservation of flow, *i.e.*, the sum of the flow coming into a basic block is equal to the sum of the flow going out of it. In order for the control-flow graph to satisfy this property, it must represent all possible control paths. Consequently, a virtual "exit" block is added so that all exits are explicitly represented as edges to the exit block. Entry to the procedure is explicitly represented by an edge from the exit block to the entry block, and the weight of this edge represents the number of times the procedure is invoked.

Because the augmented control-flow graph satisfies the conservation of flow property, it is not necessary to instrument all the edges. Instead, the weights for many of the edges can be computed arithmetically from the weights of the other edges, provided that the uninstrumented edges do not form a possibly undirected cycle. The largest cycle-free set of edges is a spanning tree. If there are $B$ blocks, there are $B-1$ edges in a spanning tree.

Any spanning tree can be used to determine a maximal set of edges that need not be measured. The sum of the weights of the tree's edges represents the savings in measurement at run time. Because a maximal spanning tree maximizes this savings, it determines optimal counter placement by keeping counters off the most frequently executed edges.

Since the edge from the exit block to the entry block does not correspond to an actual instruction in the procedure, it cannot be instrumented. Consequently, the maximal spanning tree algorithm is seeded with this edge so that it is never instrumented. The resulting spanning tree is still maximal. To see why, suppose there is a maximal spanning tree that does not contain this edge. There is only one path in this tree from the exit block to the entry block, so it must pass through exactly one of the exit block's incoming edges. By conservation of flow, the weight of this incoming edge is no greater than the weight of the single outgoing edge connecting the exit block to the entry block. Consequently, the spanning tree formed by removing this incoming edge and adding the outgoing edge has at least the same weight as the original maximal spanning tree, so it is also maximal.

Consider the *remq* function in Figure 3. It takes an element and a list, and it returns a copy of the list with all occurrences of the element removed. The function is defined using three cases. First, if the list is empty, *remq* returns the empty list. Second, if the list starts with the element, *remq* calls itself to return the result of removing the element from the remainder of the list. Otherwise, *remq* recursively calls itself to remove the element from the remainder of the list and then adds the first element to the result.

To the right of the source code in Figure 3 is a trace of *remq* removing b from the list (b a b c d). The vertical bars represent the depth of the stack. Because tail calls (such as the call to *remq* in the second condition) are essentially jumps, they do not increase stack depth.

```
(define remq
  (lambda (x ls)
    (cond
     [(null? ls) '()]
     [(eq? (car ls) x) (remq x (cdr ls))]
     [else (cons (car ls) (remq x (cdr ls)))])))
```

```
> (remq 'b '(b a b c d))
| (remq b (b a b c d))
| (remq b (a b c d))
| | (remq b (b c d))
| | (remq b (c d))
| | | (remq b (d))
| | | | (remq b ())
| | | | ()
| | | (d)
| | (c d)
| (a c d)
(a c d)
```

| | | | |
|---|---|---|---|
| | bne | arg2, *empty-list*, L1 | B1 |
| | ld | ac, *empty-list* | B2 |
| | jmp | ret | |
| L1: | ld | ac, *car-offset*(arg2) | B3 |
| | bne | ac, arg1, L2 | |
| | ld | arg2, *cdr-offset*(arg2) | B4 |
| | jmp | reloc *remq* | |
| L2: | st | arg2, 4(fp) | B5 |
| | st | ret, 0(fp) | |
| | ld | arg2, *cdr-offset*(arg2) | |
| | ld | ret, L3 | |
| | add | fp, 8, fp | |
| | jmp | reloc *remq* | |
| L3: | sub | fp, 8, fp | B6 |
| | ld | arg2, 4(fp) | |
| | ld | ret, 0(fp) | |
| | ld | arg1, ac | |
| | ld | ac, *car-offset*(arg2) | |
| | sub | ap, *car-offset*, xp | |
| | add | ap, 8, ap | |
| | st | ac, *car-offset*(xp) | |
| | st | arg1, *cdr-offset*(xp) | |
| | ld | ac, xp | |
| | jmp | ret | |

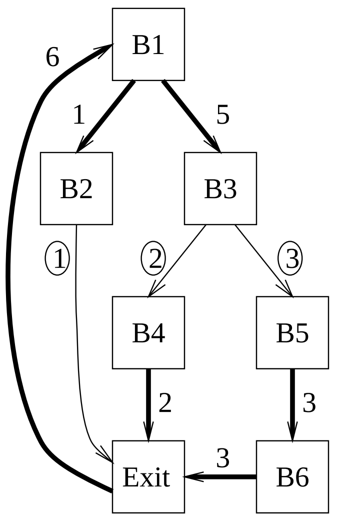


Figure 3: *remq* source, sample trace, basic blocks, and control-flow graph with thick, uninstrumented edges from one of the twelve maximal spanning trees and encircled counts for the remaining, instrumented edges

(**lambda** (*x*)
  (*f* (**if** (*P x*) *A B*) (**if** (*Q x*) *C D*)))

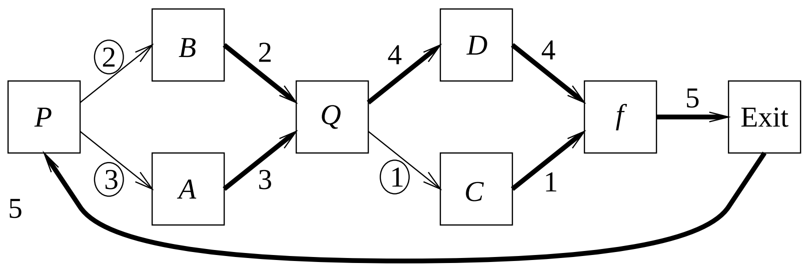


Figure 4: A loop-free Scheme function whose control-flow graph requires more than one count per entry

Below the source code in Figure 3 is sample assembly code for *remq* with run-time checks for conditions such as stack and heap overflow eliminated for simplicity of presentation. The addresses for the jumps to *remq* in B4 and B5 are filled in at link time and whenever the garbage collector relocates the code object. The assembly code is divided into basic blocks, and the corresponding control-flow graph is shown on the right. For the purposes of the graph, the tail call to *remq* in B4 is considered a jump to the exit block instead of a jump to B1, because it produces a separate invocation of *remq*. The graph assumes that each nontail call to *remq* in B5 returns exactly once to B6. In Section 3.2 we discuss what happens when this assumption fails.

There are twelve maximal spanning trees with weights summing to 20 for the graph in Figure 3, and one of these is shown by thick edges. The remaining three edges are the only ones that need to be measured at run time, and their weights are encircled in the diagram. In this particular case, there is only one counter increment per entry, the ideal situation. More complicated control-flow graphs, however, may require multiple increments per entry, even when there are no loops, as seen in Figure 4. Nonetheless, the maximal spanning tree method guarantees the minimal profiling overhead for all graphs.

## 3.2 Control-Flow Aberrations

The conservation of flow property is met only under the assumption that all procedure calls return exactly once per call. First-class continuations, however, can cause some procedure activations to exit prematurely and others to be reinstated one or more times. Even when there are no continuations, reinstrumenting a procedure while it has activations on the stack is problematic because some of its calls have not yet returned.

Scheme's *call-with-current-continuation* (*call/cc*), for example, encapsulates the current continuation as a Scheme procedure. If the continuation is invoked, the rest of the current computation is aborted, and control is resumed at the point where *call/cc* was invoked. If the continuation is invoked beyond its dynamic extent, resuming the computation involves reinstating procedure activations that already exited.

In order to demonstrate the effects of control-flow aberrations, we use a modified version of the factorial function that calls a procedure to determine the value of the base case. Figure 5 gives the Scheme source code and corresponding assembly code partitioned into basic blocks. The *fact* function is the factorial function when *done* is the identity function, (**lambda** $(x)$ $x$).

Ball and Larus handle the restricted class of exit-only continuations such as those created by *setjmp* in C by adding to each call block an edge pointing to the exit block. The weight of an exit edge represents the number of times its associated call exits prematurely. Ball and Larus measure the weights of exit edges directly by modifying continuation invocation (actually, *longjmp* and *exit*) to account for aborted activations. We use a similar strategy, but to avoid the linear stack walk overhead, we measure the weights indirectly by ensuring that exit edges are on the spanning tree (see Section 3.3).

Figure 6 shows how *call/cc* can be used to cause nonlocal exit from *fact*. As before,

```
(define fact
  (lambda (n done)
    (if (< n 2)
        (done 1)
        (* n (fact (- n 1) done)))))
```

| | | | |
|---|---|---|---|
| | bge | arg1, 2, L1 | B1 |
| | ld | cp, arg2 | B2 |
| | ld | arg1, 1 | |
| | jmp | *entry-offset*(cp) | |
| L1: | st | arg1, 4(fp) | B3 |
| | st | ret, 0(fp) | |
| | sub | arg1, 1, arg1 | |
| | add | fp, 8, fp | |
| | ld | ret, L2 | |
| | jmp | reloc *fact* | |
| L2: | sub | fp, 8, fp | B4 |
| | ld | arg1, 4(fp) | |
| | ld | ret, 0(fp) | |
| | ld | arg2, ac | |
| | jmp | reloc $*$ | |

Figure 5: Source and basic blocks for *fact*, a program demonstrating the effects of nonlocal exit and re-entry on edge counts

```
> (call/cc
    (lambda (k)
      (fact 5 k)))
| (fact 5 #<proc>)
| | (fact 4 #<proc>)
| | | (fact 3 #<proc>)
| | | | (fact 2 #<proc>)
| | | | | (fact 1 #<proc>)
1
```

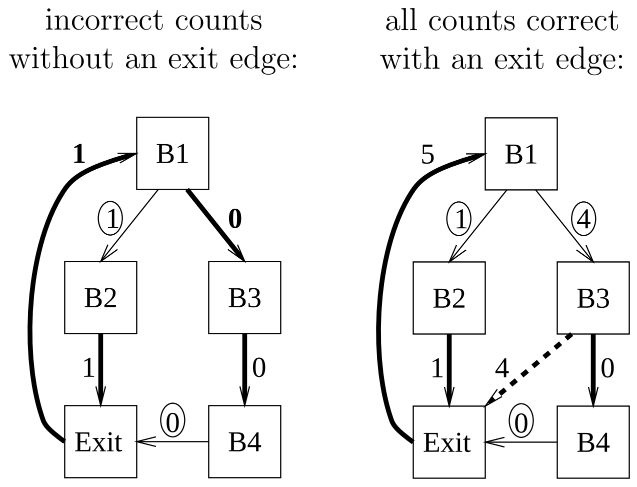


Figure 6: Trace and control-flow graphs illustrating nonlocal exit from *fact* with and without an exit edge

thick lines denote edges on a maximal spanning tree and encircled weights denote measured weights. When *call/cc* is invoked, it creates a procedure representing the continuation at that point, which is to print the result as part of the read-eval-print loop. This continuation is passed to *fact* along with five. When *done* is invoked, there are five activations of *fact* on the stack. Calling (*done* 1) aborts these activations and passes the result, one, to the printer.

Since control does not return from block B3 to block B4, the counter from block B4 to the exit block is not incremented. For the control-flow graph without an exit edge, unmeasured weight propagation results in two incorrect weights (shown in bold). The control-flow graph with an implicitly counted exit edge (shown as a dotted line) correctly accounts for this nonlocal exit, because an additional counter is placed on the edge from block B1 to block B3.

With fully general continuations, the weight of an exit edge represents the net number of times the call does not return. When a single call returns multiple times, this weight becomes negative. For example, if a certain procedure call is made and returns normally three times, but it also returns two additional times because of reinstated activations, the weight of the exit edge would be $-2$. The weight would also be $-2$ if the procedure returned five times because of reinstated activations but never returned normally because of aborted activations.

Figure 7 shows how *call/cc* can be used to cause re-entry into *fact* activations that already exited. Initially, *fact* is called with a function that uses *call/cc* to capture the continuation at the base case, save it in the global variable *redo*, and return with its input. Thus, the call to *fact* returns 4!, or 24. In the process, *redo* gets set to a continuation that multiplies its input by two, three, and four, and then prints the result. Consequently, when (*redo* 2) is called, three activations of *fact* are reinstated to compute the result, 48. (In the trace, the line "| | | | 2" represents the new result of the call to *done*, but there is no corresponding reinstated *fact* activation because

```
> (fact 4
    (lambda (n)
      (call/cc
        (lambda (k)
          (set! redo k)
          (k n)))))
| (fact 4 #<proc>)
| | (fact 3 #<proc>)
| | | (fact 2 #<proc>)
| | | | (fact 1 #<proc>)
| | | | 1
| | | 2
| | 6
| 24
24
> (redo 2)
| | | | 2
| | | 4
| | 12
| 48
48
```

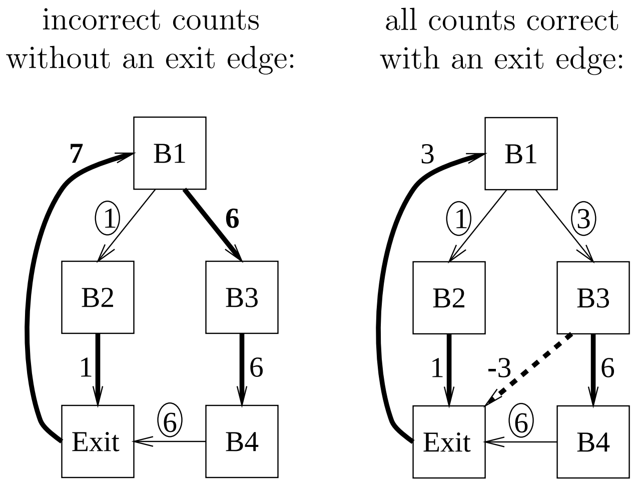


Figure 7: Trace and control-flow graphs illustrating re-entry into *fact* with and without an exit edge

the call to *done* is a tail call.)

Since control returns from block B3 to block B4 twice as many times as it enters B3, the counter from block B4 to the exit block is incremented six times. For the control-flow graph without an exit edge, unmeasured weight propagation results in two incorrect weights as before. The control-flow graph with an implicitly counted exit edge (the thick lines denote edges on a spanning tree maximal among those containing the exit edge) correctly accounts for the re-entries, because there is an additional counter placed on the edge from block B1 to block B3.

## 3.3 Implementation

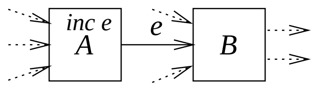
In order to support edge-count profiling in our compiler, we added a separate pass to manipulate a general-purpose symbolic assembly language common to all supported architectures. Before this addition, our compiler generated machine code directly from the final transformation pass using a machine-dependent vector of general-purpose assembly language generators. These generators include operations such as move, add, subtract, compare, and branch. They manipulate data using register, register displacement, index, immediate, and special relocation addressing modes. The new pass collects symbolic assembly code and splits it into basic blocks. Machine code is then generated from the blocks.

Our system implements procedures of variable arity using **case-lambda** [12], a variant of **lambda*** [22]. Consequently, there may be multiple entry points to a given procedure. When the compiler determines that a given call site refers to a particular case of some procedure, it generates a direct call to the appropriate entry point. To account for multiple entry points, we add a "start" block to the control-flow graph as a single virtual entry point to which all the actual entry points connect. Alternatively, we could add edges from the exit block to all the entry blocks.

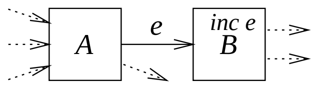
For efficiency, our compiler uses the priority-first search algorithm for finding a maximal spanning tree [38]. Its worst-case behavior is $O((E + B) \log B)$, where $B$ is the number of blocks and $E$ is the number of edges. Since each block has no more than two outgoing edges, $E$ is $O(B)$. Consequently, the priority-first algorithm performs very well with a worst-case behavior of $O(B \log B)$.

Another benefit of this algorithm is that it adds uninstrumented edges to the tree in precisely the reverse order for which their weights need to be computed using the conservation of flow property. As a result, the propagation of weights from instrumented to uninstrumented edges does not require a separate depth-first search as

If $A$'s only outgoing edge is $e$, the increment code is placed in $A$.



If $B$'s only incoming edge is $e$ (and $B$ is not the exit block), the increment code is placed in $B$.



Otherwise, the increment code is placed in a new block $C$ that is spliced into the control-flow graph.

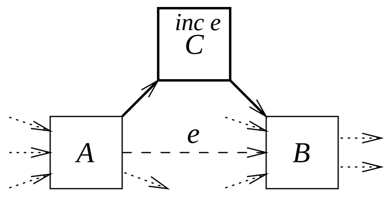


Figure 8: Efficient instrumentation of edge $e$ from block $A$ to block $B$

described by Ball and Larus [4]. Instead, the maximal spanning tree algorithm generates the list used to propagate the weights quickly and easily. This list is especially important to the garbage collector, which must propagate the weights of recompiled procedures (see Sections 4.2 and 4.4).

Figure 8 illustrates how our compiler efficiently instruments edges. It minimizes the number of additional blocks needed to increment counters by placing as many increments as possible in existing blocks. The increment instructions refer to the edge data structures by actual address.

Instrumenting exit edges is more difficult because there are no branches in the procedure associated with them. We solved this problem for the edge from the exit block to the entry block by seeding the maximal spanning tree algorithm with this edge and proving that the resulting tree is still maximal. Unfortunately, exit edges rarely lie on a maximal spanning tree because their weights are usually zero. Consequently, there are two choices for measuring the weights of exit edges.

First, we could modify the implementation of continuation invocation to update the weights of exit edges directly. Ball and Larus use this technique for exit-only continuations by incrementing the weights of the exit edges associated with each activation that would exit prematurely [4]. This approach would support fully general continuations if it would also decrement the weights of the exit edges associated with each activation that would be reinstated. The pointer to the exit edge would have to be stored either in each activation record or in a static location associated with the return address.

Second, we could make sure that all exit edges are uninstrumented by seeding the maximal spanning tree algorithm with them. In general, the resulting spanning tree would not be maximal, but it would be maximal among spanning trees that include all the exit edges. This approach is simple to implement, requiring no change to the implementation of continuations. Because the notion of maximal is weakened, however, the profiling overhead increases.

Our system's segmented stack implementation supports constant-time continuation operations [28, 8]. Implementing the first approach would destroy this property. Moreover, if any procedure is profiled, the system must traverse all activations to find the profiled ones. Although the second approach increases profiling overhead, it affects only profiled procedures, the overhead is still reasonable, and the programmer may turn off accurate continuation profiling when it is not needed. Consequently, we implemented only the second approach.

Representing nontail procedure calls in the graph is complicated by our implementation of multiple return values and run-time error checking. As shown in Figure 13, four words of data are placed in the instruction stream immediately before the single-value return point from each nontail call [28, 3, 8]. The first three words are used to support efficient garbage collection and continuation invocation. The last word, the multiple-value return address, is used when a procedure returns more than one value or zero values.

When the compiler cannot prove that the callee is a procedure, it generates code to test the callee at run time. If the test fails, the error procedure is called instead

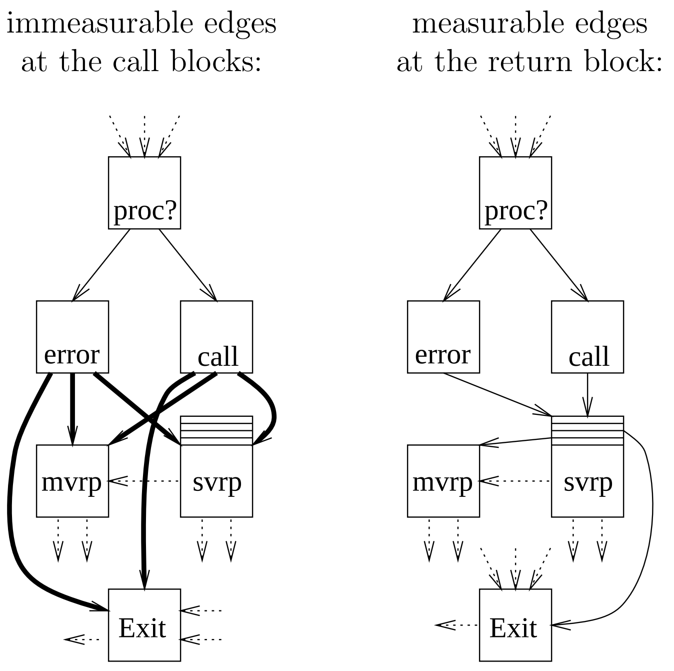


Figure 9: Control-flow graph segments showing a checked nontail procedure call with two placement strategies for the exit and multiple-value return edges

of the callee. For efficiency, one return point is shared by the call to the callee and the call to error, as shown in Figure 9. Ideally, each call block would have its own exit and multiple-value return edges, but the single- and multiple-value return edges would be impossible to instrument without replicating the return point.

Rather than replicating the return point and complicating the recompilation process to account for it when instrumentation is turned on and off, we associate one exit edge and one multiple-value return edge with the return block. As before, the exit edge is counted indirectly by seeding the maximal spanning tree algorithm with it. Its weight represents the net nonlocal exit count from both call sites. To instrument the edge from the return-point header to the multiple-value return block, the compiler simply changes the multiple-value return address to point to a new block that increments the weight of this edge. Its weight represents the total number of multiple-value returns from both call sites. Because each call block now has just one outgoing edge, these edges are instrumented by placing increments in the call blocks. The weight of each of these edges represents the number of times the corresponding call is made. The total number of single-value returns for both calls can be computed by subtracting the exit and multiple-value return counts from the sum of the two call counts.

After the compiler constructs the control-flow graph from the symbolic assembly code, it builds a spanning tree maximal among those seeded with the edges from the exit block to all the return points. In the process it builds a list of edge/block pairs that is used to propagate the values of measured edges to the unmeasured ones using the conservation of flow property. The edges not on the tree are instrumented with increments placed in existing blocks whenever possible to minimize the number of additional blocks used to increment weights.

## 3.4 Static Edge-Count Estimator

When a procedure is compiled for the first time, no profile data is available. Consequently, the edge counts are all zero, so the maximal spanning tree algorithm finds a spanning tree that may be far from maximal when the procedure actually executes. Our static estimator significantly improves initial counter placement by simulating profile data. It correctly predicts the behavior of common run-time tests such as heap and stack overflow, identifies and predicts internal loops, and assumes all other tests have a 50% chance of succeeding. Unlike Ball and Larus's estimator [4], it is linear and does not require the control-flow graph to be reducible.

The key idea that enables the estimator to be linear is that a static weighting of the control-flow graph need not satisfy the conservation of flow property. The counter placement algorithm, based on a maximal spanning tree, does not depend on this property. Only weight propagation depends on this property. Because the estimated weights are cleared before the procedure actually executes, however, they do not interfere with weight propagation.

The algorithm performs a depth-first traversal of the control-flow graph to identify and weight the back edges (internal loops) and determine a block ordering used to propagate the weights. Using this block ordering, the algorithm then processes each block, computing the total of the incoming flow and dividing it among the outgoing edges, excluding the exit and multiple-value return edges at call sites. When the algorithm recognizes a block that performs a common run-time test, it divides the flow to favor the outgoing edge that is more likely to be executed. Otherwise, it divides the flow evenly.

Figure 10 gives Scheme code for the algorithm. Each block has a *status* field that is initially unvisited. The depth-first traversal is accomplished by the *process-edge* procedure, which begins with the edge from the exit block to the start block. To prevent the algorithm from identifying this edge as a back edge, we mark the exit block visited.

When *process-edge* traverses an edge, it examines the block to which the edge points. If its status is unvisited, it changes it to active so that it can detect back edges later on, and it recursively traverses the outgoing edges (excluding exit and multiple-value return edges). The block is then marked visited and added to the front of the *order* list.

If the block's status is active, the edge is a back edge, indicating an internal loop. It is assigned the constant weight loop-weight (in our implementation, one half of top-weight), and nothing more is done. Otherwise, the block's status must be visited, and there is nothing to do in this case.

When *process-edge* finishes, all back edges are assigned a weight, and *order* contains a list of the blocks in an order appropriate for *propagate-flow*. Next, the edge

```
(define (weight-graph graph exit-block)

  (define order '())

  (define (process-edge edge)
    (let ([block (edge-to edge)])
      (cond
       [(eq? (status block) 'unvisited)
        (set-status! block 'active)
        (for-each process-edge (block-out-edges block))
        (set-status! block 'visited)
        (set! order (cons block order))]
       [(eq? status 'active) ; back edge detected
        (set-weight! edge loop-weight)]
       [else (void)]))) ; already visited

  (define (propagate-flow block)
    (let ([flow (add-weights (block-in-edges block))]
          [edges (block-out-edges block)])
      (cond
       [(usually-fails? block)
        (let ([n (quotient flow 10)])
          (set-weight! (car edges) n)
          (set-weight! (cadr edges) (− flow n)))]
       [(usually-succeeds? block)
        (let ([n (quotient flow 10)])
          (set-weight! (car edges) (− flow n))
          (set-weight! (cadr edges) n))]
       [else (divide-evenly flow edges)])))

  (let ([top-edge (car (block-out-edges exit-block))])
    (set-status! exit-block 'visited)
    (process-edge top-edge)
    (set-weight! top-edge top-weight)
    (for-each propagate-flow order)))
```

Figure 10: Scheme code for the static edge-count estimator

from the exit block to the start block is assigned the constant weight top-weight (in our implementation, 1000). Then, for each block on the list, *propagate-flow* is called to divide the incoming flow among the outgoing edges.

In our system, *propagate-flow* recognizes common run-time checks for synchronous interrupts, heap overflow, stack overflow, calling a non-procedure, incorrect number of arguments to anonymous procedures, and overflow of fixed-size integer addition and subtraction. These tests, which account for between a third and a half of the tests done at run time, behave the same way at least 90% of the time, so *propagate-flow* skews the outgoing flow by a ratio of 1:9. For the remaining blocks, *propagate-flow* divides the flow evenly among the outgoing edges.

To show that *propagate-flow* processes the blocks in an order that guarantees that each block's incoming flow has already been determined, one first shows via a straightforward proof by induction that *process-edge* adds a block to the *order* list only after all the blocks that can be reached from that block following a sequence of directed edges (excluding back edges) have been added to the list. Consequently, the list exhibits the following property. Given any block on the list, there are only three possibilities for each incoming edge: it comes out of a block that occurs earlier in the list, it is a back edge, or it is the edge from the exit block to the start block. Since *propagate-flow* already assigned the weights for outgoing edges of blocks occurring earlier in the list, all the incoming edges of the given block have valid weights. As a result, the incoming flow for each block depends only on weights that have already been computed or assigned. Moreover, when there are no back edges, this algorithm produces a weighting that satisfies the conservation of flow property.

The reason cyclic graphs usually violate the conservation of flow property is that *propagate-flow* reassigns the weights of back edges as it propagates the flow out of the blocks from which they point. An iterative update process would be required to regain the conservation of flow property, but such iterating would cause the estimator

```
(lambda (n)
  (let loop ([x (I n)])
    (if (and (P x) (Q x))
        (loop (F x))
        x)))
```

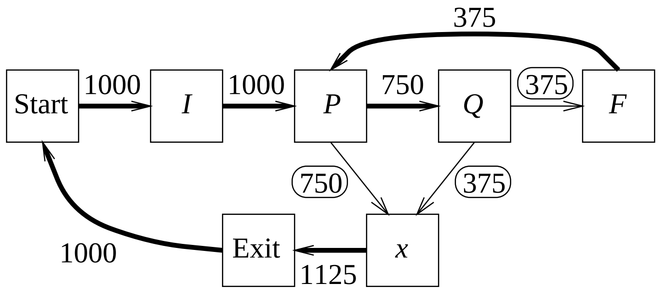


Figure 11: A Scheme function with an internal loop and its associated control-flow graph weighted by the static edge-count estimator with top-weight 1000 and loop-weight 500

to become non-linear.

The depth-first traversal touches every edge once, so it is linear with respect to the number of edges. As shown in Section 3.1, the number of edges is no more than twice the number of blocks, so the depth-first traversal is linear with respect to the number of blocks. The propagation phase touches every edge twice (from the block on each end), so it is linear, too.

Consider the Scheme function in Figure 11. The named **let** in this example expands into a **letrec** expression that binds *loop* to a separate function that processes the body of the loop. Instead of compiling it as a separate function, our compiler converts this particular type of **letrec** expression into an actual loop in assembly code. The resulting control-flow graph and the weights computed by the static edge-count estimator are also shown in Figure 11.

This example demonstrates a cyclic graph for which the estimator produces a weighting that violates the conservation of flow property at the $P$ and exit blocks but nonetheless results in optimal counter placement. The depth-first traversal of the graph identified the back edge from $F$ to $P$ and assigned it the weight 500. The subsequent propagation of flow reassigned it the weight 375. Even though this weighting violates the conservation of flow property, the maximal spanning tree determined from it turns out to be one of the two spanning trees that are maximal regardless of actual execution counts. Without a static weighting of this graph, the maximal spanning tree algorithm would find an arbitrary one of the 29 spanning trees, giving it a low chance of finding one of the two maximal ones. Unfortunately, the estimator's loop prediction algorithm is too simplistic to produce optimal counter placement for many other cyclic graphs, although it does often improve initial counter placement.

Because the estimator accurately predicts the behavior of common run-time tests and produces a reasonably flow-consistent weighting, it significantly improves initial counter placement. Moreover, it is fast, increasing average compile time by only one percent. Statistics of its speed and effectiveness in optimizing counter placement are given in Section 3.6, and statistics of its effectiveness in predicting branch behavior are given in Section 4.6.

## 3.5 Graphical Feedback

Conventional profiling tools use procedure names and line numbers to associate the data with the original source. Scheme procedures, however, do not always have names, so the compiler uses a source record that uniquely identifies the source expression. The current implementation uses the file name and the byte offset from the beginning of the file to identify the start of the source expression.

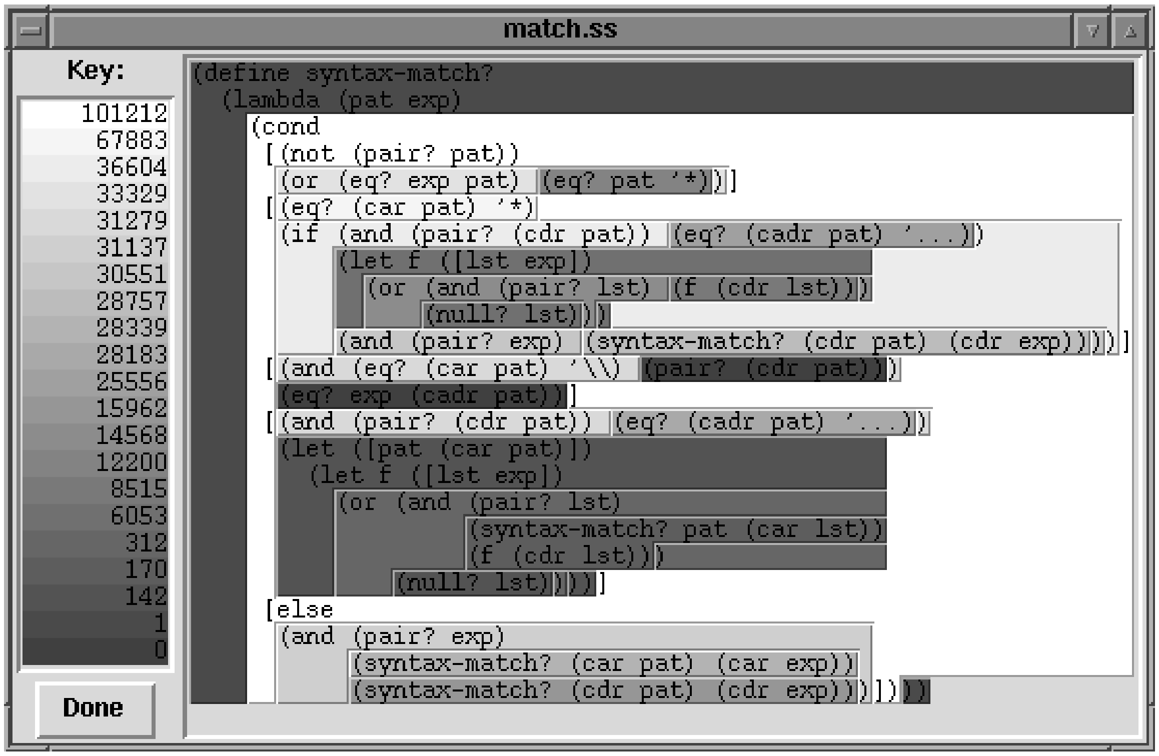
By propagating these source records through all the intermediate passes, the compiler can associate each source record with an appropriate basic block. In our system, the reader annotates its output with source records that are preserved through macro expansion by the **syntax-case** macro expander [24]. Each intermediate representation contains a source field that the transformation passes maintain. The last transformation pass generates a special "source" pseudo-instruction that contains the source record. This instruction is stored in the appropriate basic block, but it does not generate any machine code.

The normal location for a Scheme expression's source record is in the basic block that begins to evaluate it. For example, **if**, **set!**, **let**, and **letrec** are handled in this way. For procedure calls, however, the source expression corresponds to the basic block that makes the call. In most situations the count for this block is the same as the count for the block that begins to evaluate the entire call expression. A difference arises when continuations are involved, and in this case it is useful to know how many times the call is actually made.

Open-coded primitives are handled in a similar way for the same reason. Their source records are associated with the basic block that begins to evaluate the primitive after the arguments have already been evaluated.

When a constant or reference occurs in nontail position, its count can usually be determined from the count of the closest enclosing expression. An exception arises when the constant or reference occurs in nontail position as the "then" or "else" part of an **if** expression. For example, consider the program in Figure 4 when $A$, $B$, $C$, and $D$ are constants. Consequently, the compiler generates source instructions for constants and references only when they occur in tail position or as the "then" or "else" part of an **if** expression. By eliminating the source instructions for the remaining cases, the compiler significantly reduces the number of source records stored in basic blocks without sacrificing useful information.

To display the block counts in terms of the original source, we follow three steps. First, we determine the count for each block by summing the weights of all its outgoing edges. Second, we build an association list of source expressions and block counts from



```
(define syntax-match?
  (lambda (pat exp)
    (cond
     [(not (pair? pat))
      (or (eq? exp pat) (eq? pat '*))]
     [(eq? (car pat) '*)
      (if (and (pair? (cdr pat)) (eq? (cadr pat) '...))
          (let f ([lst exp])
            (or (and (pair? lst) (f (cdr lst)))
                (null? lst)))
          (and (pair? exp) (syntax-match? (cdr pat) (cdr exp))))]
     [(and (eq? (car pat) '\\) (pair? (cdr pat)))
      (eq? exp (cadr pat))]
     [(and (pair? (cdr pat)) (eq? (cadr pat) '...))
      (let ([pat (car pat)])
        (let f ([lst exp])
          (or (and (pair? lst)
                   (syntax-match? pat (car lst))
                   (f (cdr lst)))
              (null? lst))))]
     [else
      (and (pair? exp)
           (syntax-match? (car pat) (car exp))
           (syntax-match? (cdr pat) (cdr exp)))])))
```

Figure 12: A pattern matcher displayed using different colors for different execution frequencies

the source records stored in each basic block. Third, we use this list and graphical tools from the Scheme Widget Library [46] to pop up a window with the source code colored according to the counts. The programmer can also click on an expression, and the system finds and displays the corresponding count. Figure 12 gives an example.

The programmer is not limited to displaying information for one procedure at a time. Because code objects are stored in a single logical area of the heap, they can all be found at run time by a simple pass through that area. The data for all profiled procedures can be sorted by file name and then displayed using a separate window for each source file. The data can also be sorted by frequency to help programmers identify hot spots. This technique proved useful in profiling the profiler itself, helping us identify inefficiencies in the maximal spanning tree and block look-up algorithms.

## 3.6 Performance

Profiling has both run-time and compile-time costs. To assess these costs under various conditions, we used the set of Scheme benchmarks described in Table 6. All but the four largest ones came from the Gambit-C 2.3.1 benchmark suite. The measurements were taken on a DEC Alpha 3000/600 running Digital UNIX V4.0A.

Table 7 gives the results for three compilation conditions. The first compilation condition, *zero*, determines count placement by running the maximal spanning tree algorithm with all weights initially zero. The second condition, *est*, uses the static estimator to provide an initial weighting. The third condition, *prev*, uses the weights from a previous run of the benchmark. Our base of comparison is the compiler that generates and stores in the object file the control-flow graph with symbolic assembly code but does not instrument it for profiling.

The average overhead of the *zero* case is 77% at run time and 10% at compile time. The static estimator is effective at reducing the run-time overhead while increasing average compile time by just one percent. The estimator reduces the average run-time overhead by 27 percentage points and the average number of counts by 43%. Much of the benefit of the estimator comes from predicting the behavior of run-time checks that account for between a third and half of all tests done at run time. Without this prediction, the estimator results in no significant reduction in CPU time and only a 12% reduction in the number of counts.

Despite the effectiveness of the static estimator, dynamic counter placement based on weights from a previous run reduces the average run-time overhead by thirteen percentage points and the number of counts by 23%. On average, the recompiler requires 15% of the initial compile time to reinstrument and regenerate the code.

Table 8 gives the results when support for accurate profiling of control-flow aberrations is disabled. On average, removing the support decreases run time by 20% for

| *Benchmark* | *Lines* | *Description* |
|---:|---:|---|
| compiler | 35,913 | *Chez Scheme* 5.0g recompiling itself |
| softscheme | 10,073 | Andrew Wright's soft typer [50] checking his pattern matcher |
| ddd | 9,578 | Digital Design Derivation System 1.0 [7] deriving hardware for a Scheme machine [9] |
| similix | 7,305 | self-application of the Similix 5.0 partial evaluator [6] |
| nucleic | 3,475 | 3-D structure determination of a nucleic acid |
| slatex | 2,343 | SLaTeX 2.2 typesetting its own manual |
| interpret | 1,069 | Marc Feeley's Scheme interpreter evaluating takl |
| maze | 730 | Hexagonal maze maker by Olin Shivers |
| earley | 655 | Earley's parser by Marc Feeley |
| peval | 639 | Feeley's simple Scheme partial evaluator |
| boyer | 572 | Logic programming benchmark originally by Bob Boyer |
| conform | 498 | Type checker by Jim Miller |
| browse | 196 | Browse of an AI-like database of units |
| simplex | 190 | Simplex algorithm |
| puzzle | 150 | Forest Baskett's puzzle benchmark |
| trav1 | 149 | Creation of a tree structure |
| trav2 | 149 | Traversal of a tree structure |
| dderiv | 92 | Table-driven symbolic differentiation |
| fft | 77 | Fast Fourier Transform |
| destruct | 64 | Destructive operations |
| triangle | 62 | Triangle board game |
| mbrot | 51 | Generation of Mandelbrot set fractal |
| deriv | 50 | Symbolic differentiation |
| cpstak | 33 | Takeuchi function in continuation-passing style |
| ctak | 30 | Takeuchi function using *call/cc* |
| takl | 29 | Takeuchi function using lists for numbers |
| diviter | 28 | Iterative division using lists for numbers |
| divrec | 27 | Recursive division using lists for numbers |
| tak | 19 | Takeuchi function |
| fib | 18 | Recursive Fibonacci function |

Table 6: Description of profiling benchmarks

| | *Run Time* | | | *Counts* | | | *(Re)Compile Time* | | |
|---|---|---|---|---|---|---|---|---|---|
| *Benchmark* | *zero* | *est* | *prev* | *zero* | *est* | *prev* | *zero* | *est* | *prev* |
| compiler | 1.70 | 1.52 | 1.34 | 2.48 | 1.35 | 1.00 | 1.14 | 1.16 | 0.21 |
| softscheme | 1.59 | 1.51 | 1.31 | 1.81 | 1.31 | 1.00 | 1.08 | 1.10 | 0.10 |
| ddd | 1.15 | 1.12 | 1.06 | 2.29 | 1.45 | 1.00 | 1.18 | 1.19 | 0.18 |
| similix | 1.65 | 1.53 | 1.44 | 1.84 | 1.29 | 1.00 | 1.14 | 1.16 | 0.25 |
| nucleic | 1.23 | 1.18 | 1.19 | 2.08 | 1.29 | 1.00 | 1.03 | 1.03 | 0.05 |
| slatex | 1.19 | 1.09 | 1.05 | 3.53 | 1.21 | 1.00 | 1.13 | 1.13 | 0.15 |
| interpret | 2.82 | 2.25 | 1.88 | 2.05 | 1.08 | 1.00 | 1.19 | 1.19 | 0.09 |
| maze | 1.25 | 1.09 | 1.08 | 2.52 | 1.36 | 1.00 | 1.12 | 1.10 | 0.11 |
| earley | 1.35 | 1.17 | 1.16 | 2.62 | 1.17 | 1.00 | 1.08 | 1.08 | 0.13 |
| peval | 1.70 | 1.45 | 1.39 | 2.12 | 1.08 | 1.00 | 1.31 | 1.31 | 0.15 |
| boyer | 1.97 | 1.53 | 1.40 | 2.10 | 1.14 | 1.00 | 1.04 | 1.05 | 0.12 |
| conform | 2.47 | 1.95 | 1.69 | 2.31 | 1.28 | 1.00 | 1.06 | 1.10 | 0.13 |
| browse | 1.70 | 1.49 | 1.30 | 2.73 | 1.60 | 1.00 | 1.05 | 1.06 | 0.08 |
| simplex | 1.34 | 1.25 | 1.18 | 2.50 | 1.06 | 1.00 | 1.06 | 1.06 | 0.11 |
| puzzle | 1.72 | 1.42 | 1.40 | 2.16 | 1.05 | 1.00 | 1.02 | 1.04 | 0.10 |
| trav1 | 1.82 | 1.36 | 1.37 | 1.98 | 1.03 | 1.00 | 1.25 | 1.24 | 0.10 |
| trav2 | 1.88 | 1.59 | 1.43 | 1.99 | 1.49 | 1.00 | 1.03 | 1.04 | 0.05 |
| dderiv | 2.01 | 1.61 | 1.55 | 2.29 | 1.11 | 1.00 | 1.10 | 1.14 | 0.16 |
| fft | 1.06 | 1.06 | 1.05 | 1.57 | 1.10 | 1.00 | 1.08 | 1.09 | 0.15 |
| destruct | 1.77 | 1.39 | 1.28 | 2.08 | 1.31 | 1.00 | 1.10 | 1.10 | 0.16 |
| triangle | 1.89 | 1.40 | 1.37 | 3.26 | 1.97 | 1.00 | 1.02 | 1.04 | 0.13 |
| mbrot | 1.06 | 1.04 | 1.07 | 2.05 | 1.05 | 1.00 | 1.09 | 1.11 | 0.19 |
| deriv | 2.18 | 1.93 | 1.88 | 2.13 | 1.00 | 1.00 | 1.12 | 1.11 | 0.18 |
| cpstak | 1.87 | 1.56 | 1.30 | 3.00 | 1.57 | 1.00 | 1.11 | 1.11 | 0.21 |
| ctak | 1.45 | 1.38 | 1.23 | 2.54 | 1.36 | 1.00 | 1.11 | 1.11 | 0.21 |
| takl | 2.59 | 1.91 | 1.54 | 2.75 | 1.73 | 1.00 | 1.10 | 1.10 | 0.22 |
| diviter | 1.94 | 1.49 | 1.47 | 2.03 | 1.00 | 1.00 | 1.07 | 1.09 | 0.17 |
| divrec | 2.06 | 1.59 | 1.45 | 2.00 | 1.50 | 1.00 | 1.07 | 1.09 | 0.17 |
| tak | 2.47 | 2.04 | 1.65 | 2.14 | 1.57 | 1.00 | 1.09 | 1.07 | 0.25 |
| fib | 2.32 | 1.97 | 1.57 | 2.00 | 1.50 | 1.00 | 1.12 | 1.12 | 0.25 |
| *Average* | 1.77 | 1.50 | 1.37 | 2.30 | 1.30 | 1.00 | 1.10 | 1.11 | 0.15 |

Table 7: Relative costs of edge-count profiling: *zero*—instrumentation with initial weights zero, *est*—instrumentation with initial weights from the static estimator, and *prev*—instrumentation with initial weights from a previous run. The run and compile times are relative to a base compiler that does not instrument for profiling. The *prev* compile times are the recompile times relative to the base compile times. The counts are relative to those in the *prev* column.

| | *Run Time* | | | *Counts* | | | *(Re)Compile Time* | | |
|---|---|---|---|---|---|---|---|---|---|
| *Benchmark* | *zero* | *est* | *prev* | *zero* | *est* | *prev* | *zero* | *est* | *prev* |
| compiler | 1.45 | 1.46 | 1.30 | 1.34 | 1.13 | 0.76 | 1.11 | 1.12 | 0.18 |
| softscheme | 1.34 | 1.34 | 1.20 | 0.94 | 0.79 | 0.45 | 1.07 | 1.06 | 0.09 |
| ddd | 1.04 | 1.06 | 1.06 | 0.84 | 0.81 | 0.43 | 1.13 | 1.14 | 0.17 |
| similix | 1.40 | 1.40 | 1.19 | 1.00 | 0.87 | 0.53 | 1.13 | 1.12 | 0.23 |
| nucleic | 1.14 | 1.14 | 1.10 | 1.22 | 0.90 | 0.62 | 1.02 | 1.02 | 0.05 |
| slatex | 1.06 | 1.09 | 1.04 | 1.02 | 1.00 | 0.74 | 1.07 | 1.09 | 0.15 |
| interpret | 1.63 | 1.67 | 1.25 | 1.23 | 0.74 | 0.61 | 1.04 | 1.07 | 0.09 |
| maze | 1.20 | 1.19 | 1.10 | 1.25 | 1.06 | 0.66 | 1.07 | 1.07 | 0.11 |
| earley | 1.29 | 1.29 | 1.27 | 1.07 | 0.93 | 0.77 | 1.06 | 1.07 | 0.12 |
| peval | 1.43 | 1.44 | 1.29 | 1.01 | 0.87 | 0.64 | 1.30 | 1.31 | 0.13 |
| boyer | 1.51 | 1.51 | 1.34 | 1.07 | 1.03 | 0.88 | 1.02 | 1.04 | 0.11 |
| conform | 1.75 | 1.83 | 1.56 | 0.94 | 0.91 | 0.50 | 1.04 | 1.05 | 0.12 |
| browse | 1.41 | 1.47 | 1.39 | 1.14 | 1.17 | 0.69 | 1.04 | 1.04 | 0.07 |
| simplex | 1.27 | 1.19 | 1.23 | 1.46 | 1.03 | 0.96 | 1.05 | 1.05 | 0.12 |
| puzzle | 1.37 | 1.40 | 1.39 | 1.16 | 1.03 | 0.99 | 1.02 | 1.03 | 0.09 |
| trav1 | 1.32 | 1.32 | 1.34 | 0.97 | 0.96 | 0.94 | 1.21 | 1.22 | 0.10 |
| trav2 | 1.62 | 1.63 | 1.44 | 1.49 | 1.49 | 1.00 | 1.05 | 1.06 | 0.05 |
| dderiv | 1.53 | 1.54 | 1.29 | 1.12 | 0.83 | 0.67 | 1.09 | 1.13 | 0.16 |
| fft | 1.06 | 1.05 | 1.03 | 1.31 | 1.10 | 1.00 | 1.07 | 1.10 | 0.15 |
| destruct | 1.39 | 1.39 | 1.27 | 1.27 | 1.27 | 0.96 | 1.07 | 1.08 | 0.16 |
| triangle | 1.37 | 1.37 | 1.34 | 1.63 | 1.48 | 0.99 | 1.02 | 1.03 | 0.13 |
| mbrot | 1.07 | 1.03 | 1.04 | 1.03 | 1.02 | 1.00 | 1.07 | 1.10 | 0.19 |
| deriv | 1.86 | 1.78 | 1.79 | 0.92 | 0.68 | 0.68 | 1.11 | 1.09 | 0.18 |
| cpstak | 1.53 | 1.54 | 1.28 | 2.00 | 1.57 | 1.00 | 1.10 | 1.08 | 0.21 |
| ctak | 1.43 | 1.41 | 1.22 | 1.45 | 1.09 | 0.73 | 1.08 | 1.07 | 0.21 |
| takl | 1.88 | 1.87 | 1.41 | 1.65 | 1.58 | 0.83 | 1.07 | 1.10 | 0.21 |
| diviter | 1.45 | 1.49 | 1.49 | 1.02 | 1.00 | 1.00 | 1.04 | 1.06 | 0.16 |
| divrec | 1.37 | 1.37 | 1.43 | 1.00 | 1.00 | 0.50 | 1.04 | 1.07 | 0.17 |
| tak | 1.77 | 1.77 | 1.38 | 1.14 | 1.14 | 0.57 | 1.09 | 1.07 | 0.25 |
| fib | 1.65 | 1.66 | 1.24 | 1.00 | 1.00 | 0.50 | 1.10 | 1.08 | 0.21 |
| *Average* | 1.42 | 1.42 | 1.29 | 1.19 | 1.05 | 0.75 | 1.08 | 1.09 | 0.15 |

Table 8: Relative costs of profiling as in Table 7 but without support for first-class continuations and reinstrumentation of active procedures

the *zero* case, 5% for the *est* case, and 6% for the *prev* case, decreases the number of counts by 48% for the *zero* case, 19% for the *est* case, and 25% for the *prev* case, and decreases compile time by 0–2%.

The overhead of manipulating and storing symbolic assembly code is quite large. We found that the base compiler takes three times as much CPU time, allocates twice as much memory, and generates object files that are four times as large as the compiler that generates machine code directly. This overhead is still acceptable since the symbolic version compiles 315 lines of macro-intensive Scheme code per second, and the Alpha OSF/1 C compiler compiles 300 lines of C code per second. The current implementation generates machine code twice, once to determine label offsets and again to produce the actual output. Consequently, we expect that the overhead could be greatly reduced by generating machine code using a one-and-a-half pass assembler. Furthermore, each block could contain actual machine code instead of symbolic assembly code for all but the last instruction. This optimization would reduce the size of each block and eliminate the need to regenerate machine code for anything except the branches. Finally, compression of the resulting symbolic information would reduce object file size considerably.

## 3.7 Related Work

Profiling can be divided into two main areas: count-based and time-based. Ball and Larus's optimal edge-count profiling strategy [4] is the main work in count-based profiling. ATOM [42] is a system used to specify code-instrumenting tools such as profilers and can be used to implement edge-count strategies. It operates on object files and provides functions that make it easy to access procedures, basic blocks, and instructions. The current framework, however, does not allow for dynamically created procedures; all procedures must be in object files at link time.

Hall [27] describes a time-based profiling technique for functional languages that addresses the reuse of functions problem. Rather than focusing on procedures, the technique focuses on procedure calls in their full lexical context. The system uses renaming strategies to create separate instances of procedures for different lexical calling contexts. It can therefore result in exponential code growth if not carefully controlled. Hall expands the code one level at a time, rerunning a profiling suite each time to determine which calls are consuming the most resources. Only the most time consuming procedure calls are further expanded. This process can be very time consuming. It assumes the entire program text is available to the compiler at once. Because the system defines the lexical context of an anonymous procedure as the context of the closest enclosing named procedure, it does not handle anonymous and higher-order procedures very well.

Sansom and Peyton Jones [37] describe a time-based profiling technique for lazy functional languages. They allow the programmer to label any source expression with a "cost center." The cost of evaluating the labeled expression is attributed to the named cost center, which makes it easy to relate the profile data to the original source, even if there are program transformations. This technique allows programmers to work around the reuse of functions problem in that each application of a function can be attributed to a different cost center, if desired. This approach is implemented by keeping track of the current cost center and using periodic hardware timer interrupts to increment the current cost center's count. A cost center is associated with every procedure. Whenever a procedure is invoked, its cost center becomes the current cost center. Unfortunately this method does not directly allow for nested cost centers. It would be possible to have an active stack of cost centers, although this would increase the overhead of the timer interrupt routine's update process.

# Chapter 4

# Dynamic Recompilation

This chapter presents an efficient infrastructure for dynamic recompilation in Scheme using the example of basic block reordering based on edge-count profile data. The infrastructure enables completely transparent recompilation of procedures at run time, even if some of them have live activations on the stack or in captured continuations. To support dynamic recompilation, the system must be able to find all pointers to a given procedure at run time. This property is satisfied by modern garbage collectors but may also be satisfied in other ways. The infrastructure adds negligible overhead to unprofiled programs and requires only minor changes to the garbage collector.

Section 4.1 gives some background on profile-based optimizations. Section 4.2 presents an overview of dynamic recompilation in our system. Section 4.3 summarizes the representations of Scheme objects and how they are modified to support dynamic recompilation. Section 4.4 describes how the garbage collector transparently replaces code objects with recompiled ones. Section 4.5 illustrates dynamic recompilation using a variant of Pettis and Hansen's basic block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [35]. Section 4.6 reports the cost of recompilation and the effectiveness of dynamic block reordering versus static block ordering based on the edge-count estimator described in Section 3.4. Section 4.7 describes related work.

## 4.1 Background

Profiling tools have certainly been valuable for programmers by helping them identify execution bottlenecks and inefficiencies. Anderson [1] demonstrates the value of profiling with two 2000-line machine learning programs. Both are floating-point intensive, and the second is also structure manipulation intensive. Each program was written in both C and Lisp, and the C versions were "highly optimized." One day was spent tuning the Lisp versions using profile data. The first Lisp program was improved by 40%, making it slightly faster than the C version. The second Lisp program was improved by a factor of 30, making it over twice as fast as the C version.

Profiling tools can also provide useful information for compilers. Current computer architectures increase performance by predicting and prefetching instructions, by issuing multiple instructions in a single clock cycle, and by reading from and writing to large caches. Consequently, it has become increasingly important for compilers to generate code that takes factors such as instruction scheduling, branch prediction, and code locality into account. Profile data can assist the compiler is these areas.

Basic blocks can be reordered to reduce the number of mispredicted branches and instruction cache misses [35, 36]. Although there are various heuristic algorithms that estimate branch behavior statically, research has demonstrated that profile-based prediction is significantly more effective [5, 47, 34]. In current programming environments, profile-based prediction requires a tedious compile-profile-recompile cycle. In our system, the instrumentation for profiling and the subsequent recompilation are done at run time.

## 4.2 Overview

Dynamic recompilation proceeds in three phases. First, candidate procedures are identified, either by the user or by a program that selects among all the procedures in the heap. Second, these procedures are recompiled and linked to separate, new procedures. Third, the original procedures are replaced by the new ones during the next garbage collection.

Because a procedure's entry and return points may change during recompilation, the recompiler creates a translation table that associates the entry- and return-point offsets of the original and new procedures and attaches this table to the original procedure. The collector uses the translation table to relocate *direct calls* (calls to known entry points) and return addresses. Because the collector translates return addresses, procedures can be recompiled while they are executing. At the end of collection, the storage from the original procedures and translation tables is freed.

Before the next collection, only the original procedures are used. This invariant allows each original procedure to share its control-flow graph and associated counts with the new procedure. Because the new procedure's maximal spanning tree (see Chapter 3) may be different from the original's, the new procedure may increment different counts. The collector accounts for the difference by computing the unmeasured counts of the original procedure so that the new procedure starts with a complete set of accurate counts.

In environments where it is generally impossible to locate all pointers to a given procedure at run time, dynamic recompilation can be implemented by introducing one level of indirection into procedure calls and returns. For each procedure call, the compiler can generate a jump to an entry stub that in turn jumps to the actual entry point. Similarly, the compiler can cause each procedure call to return through a return stub that jumps to the appropriate return point. With this design, translating entry and return points is as simple as changing the corresponding stubs. The extra branch may not reduce performance significantly, especially in systems with a low frequency of procedure calls or with a similar mechanism already in place for dynamic linking. With branch-folding hardware, there would be no overhead, because the branches would be eliminated from the pipeline.

Dynamic recompilation need not be limited to low-level optimizations such as block reordering. By associating profile data with earlier compilation passes, the recompiler can perform run-time optimizations involving register allocation, flow anaylsis, lambda lifting, and procedure inlining. Because these optimizations destroy the one-to-one correspondence among basic blocks, the collector must be sensitive to return points in original procedures that have no corresponding, compatible return points in recompiled procedures. By retaining original procedures as long as they have live unassociated return addresses, the collector can support these optimizations—even on running procedures.

## 4.3 Object Representations

Every Scheme object consists of a type tag and a value. There are three main methods for representing the type tag [43]: associating the type tag with the object (*typed objects*), associating the type tag with all pointers to the object (*typed pointers*), and associating the tag type with the segment in which the objects reside (*big bag of pages*).

Our system uses a hybrid of these three methods, which are described in detail in [21]. Every Scheme object is represented by a 32-bit tagged integer. For a heap-allocated object, the tagged integer is a typed pointer. The primary type tag is stored in the lower three bits of the pointer. The starting address is always aligned on an even eight-byte boundary so that the lower three bits are zero. If we simply add the address and the tag to form the pointer, we would need a negative displacement to access the first element of the object. Because some architectures do not support negative displacements, the pointer is eight less than the address plus the tag.

Scheme objects that are not heap-allocated are called *immediate* objects. For efficiency, immediate objects are encoded directly in the tagged integers themselves. In order to support efficient 30-bit integer arithmetic, our system dedicates the tags 000 and 100 to *fixnums*. The upper 30 bits of the tagged integer contain the 30-bit integer. To add or subtract two fixnums, the tagged integers are added or subtracted, respectively. To multiply two fixnums, one of the tagged integers is shifted two bits to the right before being multiplied by the other tagged integer. To divide two fixnums, the tagged integers are divided, and the quotient is shifted to the left two bits. These additional shifts are avoided when one of the arguments is constant.

Other immediate objects, such as characters, booleans, and the empty list, are encoded in the upper 29 bits of the tagged integer and are given the tag 110. The tag 111 is used as an escape code for typed objects, which include vectors, strings, and code objects. The four remaining tags are used for pairs, floating-point numbers, symbols, and closures.

In order to describe the infrastructure for dynamic recompilation, we first show how Scheme procedures are represented without the infrastructure. Then we explain how some of the unused bits of the representations are used to support the infrastructure efficiently.

Scheme procedures are represented as closures and code objects. In general, there is one code object for each procedure (**lambda** expression), and a closure is created each time a procedure is evaluated. Consequently, we use the generic term *procedure* to refer to code objects, not closures. A closure is a variable-length array whose first element contains the address of the procedure's generic entry point and whose remaining elements contain the procedure's free variables, as shown in Figure 13. The

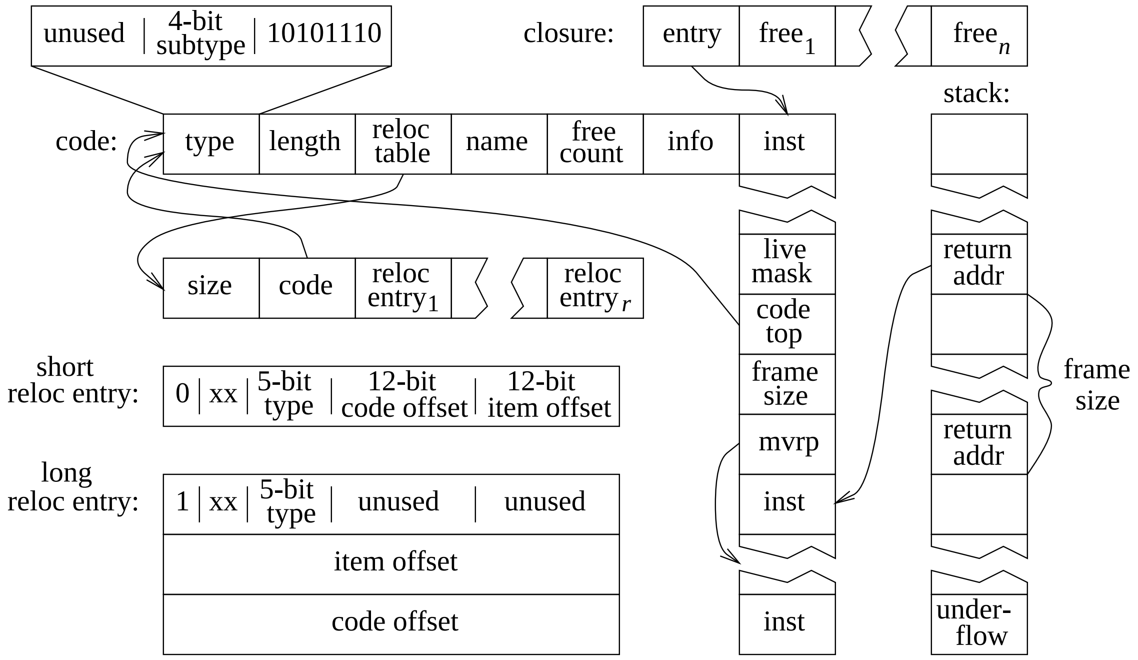


Figure 13: Representations of closures, code objects, and stack segments without the infrastructure for dynamic recompilation

number of free variables is stored in the code object associated with the generic entry point. If the $i^{\text{th}}$ free variable is not assigned, its value is stored in free$_i$; otherwise, a tagged pointer to a heap-allocated cell containing its value is stored there.

The generic entry point is the first instruction of the procedure's machine code, which is stored in a code object. A code object has a six-word header before the machine code, as illustrated in Figure 13. Since a code object is a typed object, the first word contains the type tag, which includes a four-bit subtype. The second word contains the length of the machine code in bytes. The third word contains the address of a relocation table, described below. The fourth word contains the name of the procedure, either a string pointer or #f. The fifth word contains the number of free variables. The sixth word contains a pointer to additional information used by the debugger and recompiler.

A relocation table is used by the garbage collector to relocate items stored in the instruction stream. The first element contains the number of relocation entries in the table. The second element contains a pointer to the code object, needed during garbage collection to relocate relative addresses. The subsequent elements contain relocation entries that come in two sizes. The high-order bit indicates whether the entry is short or long. A relocation entry specifies the offset of the item within the code stream (the code offset), the offset from the item's address to the address actually stored in the code stream (the item offset), and how the item is encoded in the code stream (the type). The two x's in the figure indicate unused bits.

In order to support multiple return values, first-class continuations, and garbage collection efficiently, four words of data are placed in the instruction stream immediately before the single-value return point from each non-tail call [28, 3, 8]. The first is a live mask, a bit vector describing which frame locations contain live data, used by the garbage collector. The second is a pointer to the top of the code object, and the collector uses it to find the code object associated with a given return address in a stack segment. The third is the size of the frame, and it is used during collection and continuation invocation to walk down a stack segment. The fourth is the multiple-value return address used when a procedure returns more than one value or zero values.

There are several important considerations in the design of the infrastructure for dynamic recompilation. Suppose a given code object has been recompiled. The garbage collector must then relocate all references to the original code object so that they point to the new one. The original code object must therefore be marked with a pointer to the new one.

Because the machine code in the new code object may be quite different from the machine code in the original code object, a translation table is needed to convert references to particular instructions in the original code object to the corresponding instructions in the new one. These types of references occur only as return addresses in stack segments and direct calls in code objects. Consequently, the translation table must contain an entry for each return point and entry point (except the generic entry point, if any, which is always the first instruction).

Suppose a short relocation entry contains a direct call to a recompiled code object. The 12-bit item offset is the offset of the entry point in the original code object, but the offset of the corresponding entry point in the new code object need not fit in the 12-bit field. Consequently, the relocation entries for direct calls use the long format to avoid resizing of the relocation table. A side benefit of this approach is that the long format provides a convenient location to store a bit indicating that the entry corresponds to a direct call. Use of the long format does not significantly increase the table size because direct calls account for a minority of relocation entries. Because short entries cannot describe direct calls, they need not be checked during relocation.

Figure 14 highlights the modifications to the representation of code objects to support dynamic recompilation. The first word of a long-format relocation entry reserves the low-order "d" bit to indicate whether or not the given entry corresponds to a direct call.

Three of the unused type bits are used to encode the status of a code object. The first bit, set by the recompiler, indicates whether or not the code object has been recompiled to a new one and is thus *obsolete*. Since an obsolete code object will not be copied during collection, its relocation table is no longer needed. Therefore, its relocation field is used instead to point to the translation table. The translation table contains a pointer to the new code object and a list of original/new offset pairs. The pairs are sorted by original offset to enable fast binary searching during relocation.

The second bit, denoted by "n" in the figure, indicates whether the code object is *new*. This bit, set by the recompiler and cleared by the garbage collector, is used to prevent a new code object from being recompiled before the associated obsolete one has been eliminated. The third bit, denoted by "b" in the figure, serves a similar

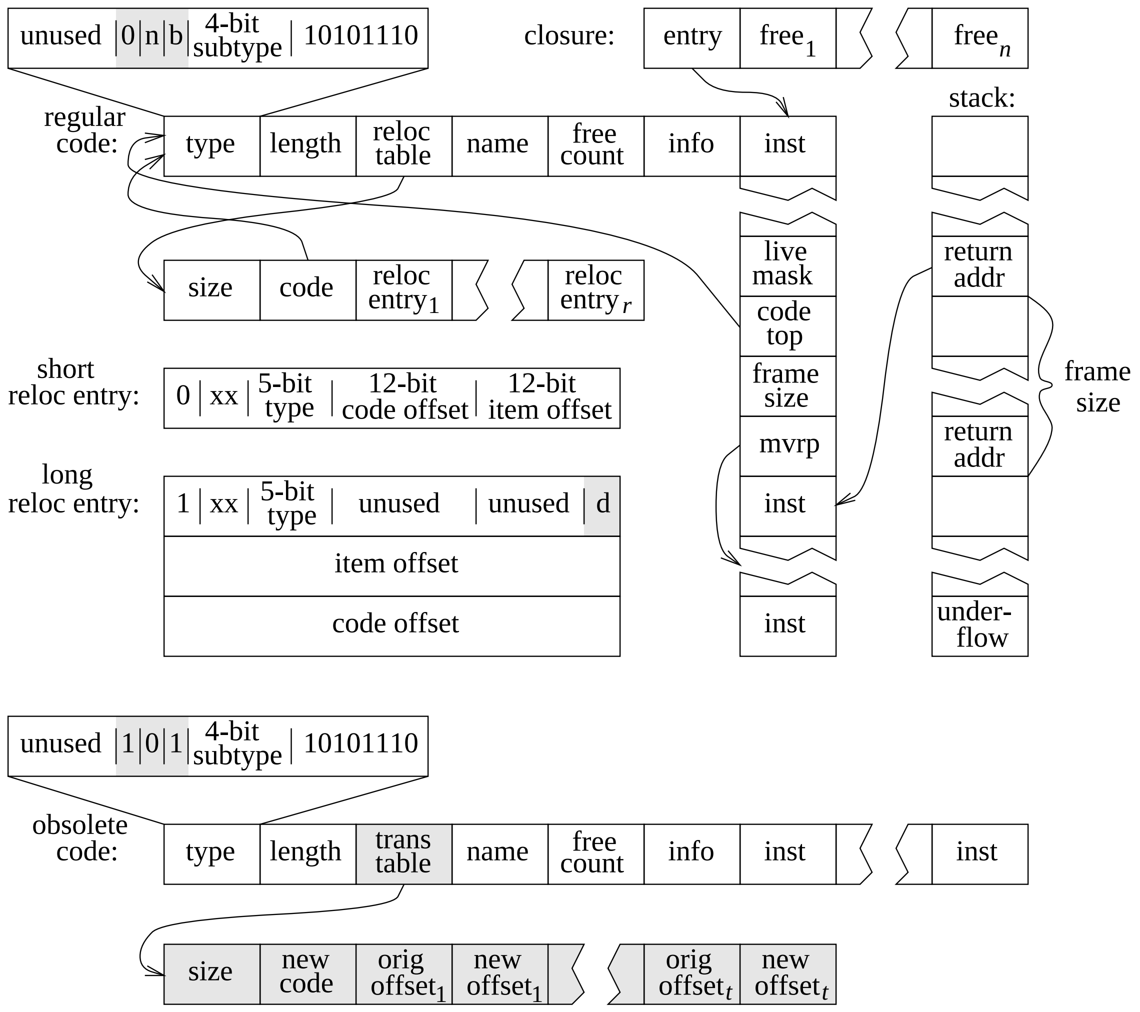


Figure 14: Representations of closures, code objects, and stacks with the infrastructure for dynamic recompilation highlighted

purpose. It indicates whether the code object is *busy* in that it is either being or has been recompiled. This bit is used to prevent multiple recompilations of the same code object. The recompiler sets this bit at the beginning of recompilation. Because the recompiler may trigger a garbage collection before it creates a new code object and marks the original one obsolete, this bit must be preserved by the collector. It is possible to recompile a code object multiple times; however, each subsequent recompilation must occur after the code object has been recompiled and collected. It would be possible to relax this restriction, but in practice we have found no reason to do so.

## 4.4 Garbage Collection

In order to describe how our generational [32], stop-and-copy [25] garbage collector transparently replaces obsolete code objects with recompiled ones, we first summarize how it manages objects without the infrastructure for dynamic recompilation. We then explain the minor modifications necessary to support the infrastructure. (See Wilson [48] for an excellent survey of garbage collection techniques.)

Our storage management system subdivides objects logically by *generation* and *space*. Objects of the same generation have survived the same number of collections. Objects of the same space are swept the same way by the collector. Spaces include the following:

*impure*—for mutable, pointer-containing objects such as pairs and vectors
*pure*—for immutable, pointer-containing objects such as closures
*code*—for code objects only
*data*—for unswept objects such as strings
*continuation*—for continuation objects only

The logical subdivision of objects is implemented by partitioning the heap into fixed-size segments, each of which contains objects of only one generation and space. A resizable segment table associates with each segment its generation and space. Consequently, the segments comprising a given generation and space need not be contiguous. See [21] for a more detailed description of our segmented storage model.

The garbage collector takes three inputs: the root set of pointers, the oldest generation to be collected ($o$), and the target generation ($t$). All objects in generations zero to $l$ will be collected into generation $t$. Either $t = o + 1$ for normal promotion of objects, or $t = o$ for capping promotion at generation $o$. All segments whose generation is less than or equal to $l$ are marked *old* and are collectively called *old space*. All objects in old space that can be reached from the root set and from older generations are copied into generation-$t$ segments, which are collectively called *new space*. When collection finishes, all old space segments are marked free.

The collection algorithm uses a form of breadth-first search to traverse all reachable objects in old space [16]. The queue of objects for each space is maintained in new space by associating an allocation pointer (the back of the queue) and a sweep pointer (the front of the queue) with each space in generation $t$. The search begins by relocating the root set of pointers. Next, the sweep pointers are used to identify objects whose internal pointers need to be relocated. For each space except data, the sweep pointer is advanced (removing objects from the front of the queue) as internal pointers are relocated until the sweep pointer reaches the allocation pointer (the queue is empty). The different spaces use different approaches to sweep the objects, as described below.

To relocate a pointer to an unforwarded object, the collector copies the object into new space, updates the appropriate allocation pointer, and places a forwarding marker and pointer at the old location. Because the new object is logically between the appropriate sweep and allocation pointers, it is now on the queue of objects to be swept. The forwarding pointer is used to relocate other pointers to the object.

In our system, the forwarding marker is a unique immediate object, and it is placed in the first word of the old object. A tagged pointer to the copied object is placed in the second word of the old object. Because all heap allocation is double-word aligned, there is always space in the old object for the forwarding marker and pointer.

Because all the objects in the impure and pure spaces contain nothing but tagged pointers and immediates, they are swept one word at a time, independent of object boundaries. Consequently, objects that do not occupy an even number of words are padded with the fixnum zero. Closures appear to violate the tagged pointer rule because their entry fields contain actual addresses. The violation is resolved by ensuring that each code object's generic entry point is aligned on an even four-byte boundary so that its address masquerades as a fixnum during the sweep phase. Because the entry fields will not be relocated during the sweep phase, the entry fields and associated code objects are relocated when closures are copied. Recursion of the garbage collector is avoided because no pointers are relocated when code objects are copied.

Since code objects have pointers embedded in their machine code, they are swept one code object at a time. Because the only pointer to a code object's relocation table is in the relocation field, the table is stored in the data space (which is not swept) and copied when its code object is copied. Consequently, the sweep phase does not have to relocate the address in the relocation field. Instead, it uses the relocation table to relocate the pointers embedded in the machine code. When it finishes, it updates the relocation table's code object pointer.

Continuation objects are stored in their own space because they contain untagged pointers and thus must be swept one continuation object at a time. Continuation objects are represented by closures, and they are distinguished from regular closures by a bit in the type field of the code object associated with the entry field. The free variable slots are used to store information about the continuation object such as the size and address of its stack segment (see [28, 8] for a more detailed description). Since the only pointer to a continuation object's stack segment is in the continuation object, the stack segment is stored in the data space and copied when its continuation object is copied. When a continuation object is swept, its stack segment is swept one frame at a time using the live mask, code pointer, and frame size fields stored behind each return address.

Pointers from objects in generations older than $o$ to objects in younger generations are relocated by sweeping the appropriate "dirty" sections of the older generations. The dirty sections are identified using a form of card marking [41, 49]. The overhead of maintaining the marks is reduced by recognizing that only mutable objects can contain pointers to younger generations. Consequently, dirty marks are not maintained for the pure, code, data, and continuation spaces.

Each generation/space pair requires a separate allocation pointer. To improve the efficiency of allocation, the compiler generates in-line code to allocate new objects into generation zero's impure space. The garbage collector always copies an object into the appropriate space of the target generation. Consequently, the collector never sweeps objects allocated in generation zero, so there is no problem omitting the space information for new objects. The in-line allocation code, therefore, can simply increment a single allocation pointer and compare it against a single pointer to the end of the current segment. On most architectures, both of these pointers are kept in machine registers to make new allocation very fast.

Only a few modifications are necessary to support the infrastructure for dynamic recompilation. The primary modification involves how a code object is copied. If the obsolete bit is clear, the code object and associated relocation table are copied as usual, but the "n" bit of the copied code object's type field is cleared if set.

If the obsolete bit is set, the code object is not copied at all. Instead, the pointer to the new code object (found in the translation table) is relocated. The resulting pointer is stored as the forwarding address for the obsolete code object so that all other pointers to the obsolete code object will be forwarded to the new one. Moreover, if the obsolete code object is instrumented for profiling, the collector propagates the weights so that they remain accurate when the new code object increments a possibly different subset of the weights. The list of edge/block pairs used for propagation is found in the "info" structure. Since the propagation occurs during collection, some of the objects in the list may be forwarded, so the collector must check for pointers to forwarded objects as it propagates the weights.

The translation table is used to relocate direct calls and return addresses. Direct calls are found only in code objects, so they are handled when code objects are swept. The "d" bit of long-format relocation entries identifies the candidates for translation. When a direct entry refers to a code object that is obsolete, the offset of that entry is translated using the obsolete code object's translation table and updated with the new offset. Return addresses are found only in stack segments, so they are handled when continuation objects are swept. When a return address refers to an obsolete code object, the offset of the return address is translated using the obsolete code object's translation table to determine the corresponding return address in the new code object.

The relocation of pointers to obsolete code objects is optimized by placing a forwarding pointer on the obsolete code object itself after the associated new code pointer is relocated. Because the forwarding marker overwrites the type field of obsolete code objects, the type bits cannot be used to identify obsolete code objects when sweeping relocation tables and continuations. Instead, the name field is used to identify them. Since a code object's name field cannot be a fixnum, the collector sets the name field of an obsolete code object to fixnum zero just before it puts down the forwarding marker and pointer. Consequently, relocation of direct calls and return addresses involves a simple zero test of the name field instead of a bit test of the type field.

Since our collector is generational, we must address the problem of potential cross-generational pointers from obsolete to new code objects. Our segmented heap model allows us to allocate new objects in older generations when necessary [21]; thus, we always allocate a new code object in the same generation as the corresponding obsolete code object. Otherwise, we would have to add the pointer to our remembered set and promote the new code object to the older generation during collection.

## 4.5 Block Reordering Example

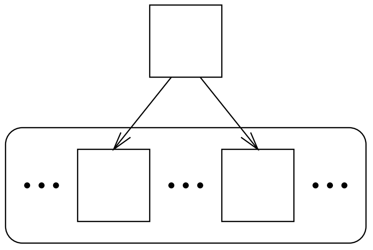
To demonstrate the feasibility and utility of dynamic recompilation, we use a variant of Pettis and Hansen's basic block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [35]. We also use edge-count profile data to decrease profiling overhead by re-running the maximal spanning tree algorithm to improve counter placement, as described in Chapter 3.

The block reordering algorithm proceeds in two steps. First, blocks are combined into chains according to the most frequently executed edges to reduce the number of instruction cache misses. Second, the chains are ordered to reduce the number of mispredicted branches.

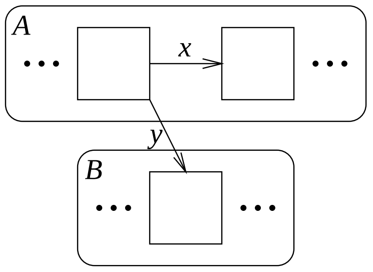
Initially, every block comprises a chain of one block. Using a list of edges sorted by decreasing weight, distinct chains $A$ and $B$ are combined when an edge's source block is at the tail of $A$ and its sink block is at the head of $B$. When all the edges have been processed, a set of chains is left.

The algorithm places ordering preferences on the chains based on the conditional branches emitted from the blocks within the chains and the target architecture's

If both edges point to blocks in the same chain, no preferences are added.



If $x \geq y$, $A$ should come before $B$ with weight $x - y$. Otherwise, $B$ should come before $A$ with weight $y - x$.



If $x \geq y$, $B$ should come before $A$ before $C$ with weight $x - y$. Otherwise, $C$ should come before $A$ before $B$ with weight $y - x$.

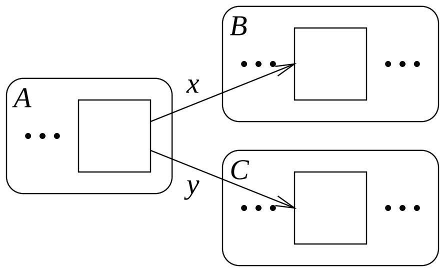


Figure 15: Adding branch prediction preferences for architectures that predict backward conditional branches taken and forward conditional branches not taken

branch prediction strategy. Blocks with two outgoing edges always generate a conditional branch for one of the edges, and they generate an unconditional branch when the other edge does not point to the next block in the chain. Figure 15 illustrates how the various conditional branch possibilities generate preferences for a common prediction strategy. The preferences are implemented using a weighted directed graph with nodes representing chains and edges representing the “should come before” relation.

As each block with two outgoing edges is processed, its preferences (if any) are added to the weights of the graph. Suppose there is an edge of weight $x$ from chain

$A$ to $B$ and an edge of weight $y$ from chain $B$ to $A$, and $x > y$. The second edge is removed, and the first edge's weight becomes $x - y$, so that there is only one positive-weighted edge between any two nodes. A depth-first search then topologically sorts the chains, omitting edges that cause cycles. The machine code for the chains is placed in a new code object, and the old code object is marked obsolete and has its relocation field changed to point to the translation table.

## 4.6 Performance

To assess the cost of recompilation using our block reordering algorithm, we measured the run time, compile time, and recompile time for the set of Scheme benchmarks described in Table 6 in Chapter 3. We also measured the number of unconditional branches (jumps) between blocks and the break-down of forward/backward, taken/not-taken conditional branches.

As before, the measurements were taken on a DEC Alpha 3000/600 running Digital UNIX V4.0A. The Alpha architecture encourages hardware and compiler implementors to predict backward conditional branches taken and forward conditional branches not taken [40]. Current Alpha implementations use this static model as a starting point for dynamic branch prediction. We computed the mispredicted branch percentage using the static model as a metric for determining the effectiveness of the block reordering algorithm.

Table 9 gives the results. The *best* column gives the static mispredicted branch percentage assuming each branch is correctly predicted, even if there is no block ordering that would cause every branch to be correctly predicted. The *dyn* columns show the effectiveness of the dynamic block reordering algorithm using profile counts from a previous run. The *stat* columns show the effectiveness of the block reordering algorithm using estimated counts from the static estimator of Section 3.4. The base

| | *% Mispredicted* | | | | *% Jumps* | | | *Run Time* | | *Comp Time* | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Benchmark* | *best* | *dyn* | *stat* | *base* | *dyn* | *stat* | *base* | *dyn* | *stat* | *dyn* | *stat* |
| compiler | 6 | 9 | 25 | 72 | 9 | 9 | 11 | 0.87 | 0.92 | 0.16 | 1.07 |
| softscheme | 7 | 7 | 45 | 38 | 1 | 2 | 3 | 0.97 | 1.09 | 0.07 | 1.02 |
| ddd | 4 | 4 | 31 | 62 | 0 | 1 | 13 | 0.98 | 1.04 | 0.12 | 1.06 |
| similix | 7 | 7 | 27 | 61 | 0 | 2 | 2 | 0.99 | 0.99 | 0.16 | 1.05 |
| nucleic | 1 | 1 | 2 | 97 | 0 | 1 | 1 | 0.92 | 0.93 | 0.04 | 1.03 |
| slatex | 2 | 3 | 26 | 77 | 4 | 14 | 27 | 0.97 | 0.98 | 0.11 | 1.01 |
| interpret | 1 | 1 | 8 | 46 | 0 | 0 | 0 | 0.90 | 1.33 | 0.06 | 0.98 |
| maze | 10 | 10 | 16 | 79 | 9 | 11 | 10 | 0.92 | 0.93 | 0.08 | 1.02 |
| earley | 20 | 24 | 29 | 65 | 15 | 15 | 19 | 0.99 | 1.02 | 0.09 | 1.02 |
| peval | 11 | 11 | 30 | 66 | 0 | 1 | 6 | 0.92 | 0.97 | 0.09 | 1.04 |
| boyer | 3 | 3 | 6 | 96 | 26 | 26 | 26 | 0.86 | 0.88 | 0.08 | 1.03 |
| conform | 4 | 4 | 11 | 76 | 2 | 14 | 4 | 0.95 | 0.96 | 0.09 | 1.04 |
| browse | 8 | 8 | 18 | 82 | 7 | 11 | 7 | 0.95 | 0.97 | 0.06 | 1.01 |
| simplex | 10 | 18 | 18 | 66 | 18 | 16 | 23 | 0.95 | 0.97 | 0.09 | 1.04 |
| puzzle | 4 | 5 | 30 | 65 | 3 | 31 | 31 | 0.95 | 0.95 | 0.08 | 1.01 |
| trav1 | 1 | 1 | 3 | 99 | 46 | 46 | 46 | 0.95 | 0.95 | 0.08 | 0.99 |
| trav2 | 1 | 1 | 1 | 67 | 1 | 33 | 33 | 0.88 | 0.99 | 0.04 | 1.06 |
| dderiv | 8 | 8 | 10 | 69 | 4 | 4 | 4 | 0.96 | 0.96 | 0.12 | 1.05 |
| fft | 3 | 6 | 11 | 84 | 6 | 13 | 12 | 0.98 | 0.99 | 0.13 | 1.06 |
| destruct | 3 | 3 | 4 | 95 | 25 | 26 | 30 | 0.91 | 0.91 | 0.13 | 1.04 |
| triangle | 6 | 6 | 37 | 63 | 0 | 0 | 16 | 0.95 | 0.98 | 0.11 | 0.99 |
| mbrot | 0 | 1 | 0 | 99 | 1 | 9 | 9 | 0.98 | 0.98 | 0.16 | 1.07 |
| deriv | 9 | 9 | 13 | 68 | 0 | 0 | 0 | 0.96 | 0.96 | 0.16 | 1.06 |
| cpstak | 6 | 6 | 6 | 65 | 0 | 0 | 0 | 0.98 | 0.98 | 0.18 | 1.09 |
| ctak | 5 | 5 | 5 | 67 | 0 | 0 | 0 | 0.98 | 1.00 | 0.17 | 1.06 |
| takl | 5 | 5 | 5 | 69 | 0 | 0 | 0 | 0.90 | 0.93 | 0.18 | 1.07 |
| diviter | 0 | 0 | 0 | 99 | 33 | 33 | 33 | 0.89 | 0.95 | 0.14 | 1.06 |
| divrec | 0 | 0 | 0 | 100 | 0 | 0 | 0 | 0.94 | 0.92 | 0.14 | 1.03 |
| tak | 11 | 11 | 11 | 67 | 0 | 0 | 0 | 1.00 | 1.00 | 0.23 | 1.04 |
| fib | 20 | 20 | 20 | 80 | 0 | 0 | 0 | 0.95 | 0.95 | 0.23 | 1.06 |
| *Average* | 6 | 7 | 15 | 75 | 7 | 11 | 12 | 0.94 | 0.98 | 0.12 | 1.04 |

Table 9: Performance of dynamic recompilation using block reordering: *best*—all conditional branches predicted correctly, *dyn*—dynamic reordering based on a previous run, *stat*—block ordering based on the static estimator, and *base*—default block ordering. The number of jumps is relative to the number of conditional branches.

of comparison for the branch statistics and the timings is the default compiler, *base*, which does not reorder basic blocks.

For simplicity of code generation, most of the common run-time checks for conditions such as heap and stack overflow were designed to test and branch around the code that handles the condition. Because these tests almost always fail, the mispredicted forward conditional branches inflate the *base* misprediction percentage. Factoring them out yields an initial misprediction rate of about 50%. Rather than using *ad hoc* code generation techniques to remedy this situation, we decided to develop the more general block reordering strategy presented here.

Combined with our fast static estimator, the block reordering algorithm is quite effective at reducing the number of mispredicted branches while also slightly decreasing the number of jumps. Our algorithms are also efficient, adding a mere 4% to the average compile time.

Although static block reordering reduces the number of mispredicted branches significantly, dynamic block reordering reduces it to a nearly minimal level and significantly reduces the number of jumps between blocks. The reordering and recompilation require only 12% of the base compile time.

Run-time reduction varies considerably among benchmarks and appears to have little correlation with the static misprediction rate. This variation is most likely caused by the microprocessor's dynamic branch prediction mechanism. These results suggest that conditional branch instructions that specify static-only prediction would be useful. Moreover, the speed and effectiveness of our system's block reordering algorithm indicate that dynamic branch prediction may not be cost-effective in the future. We hypothesize that the location of procedures in the heap also has a significant effect on run time, especially on machines with direct-mapped caches such as the Alpha. An important area of future work is to apply reordering techniques to procedures as well as basic blocks to reduce instruction cache conflicts.

## 4.7 Related Work

In the absence of profiling information, various heuristic algorithms can be used for statically estimating branch behavior. Ball and Larus [5] describe a simple heuristic that can significantly reduce the number of mispredicted branches. They found, however, that on average their static prediction generates code that executes twice as many mispredicted branches as the code generated by profile-based prediction. Wall [47] also found that profile-based prediction is significantly better than any of his heuristic-based predictions. Patterson [34] uses value range propagation to predict branch behavior. Even though his technique does better than Ball and Larus's, profile-based prediction is still substantially better.

Profiling information can be used to increase the locality of executing code at both intra- and inter-procedural levels. Section 4.5 summarizes Pettis and Hansen's [35] intraprocedural block reordering algorithm. Samples [36] explores a similar intraprocedural algorithm that reduces instruction cache miss rates by up to 50%. Pettis and Hansen [35] also describe an interprocedural algorithm that places procedures in memory such that those that execute close together in time will also be close together in memory. Since determining the optimal ordering is NP-complete, they use a greedy "closest is best" strategy.

Other optimizations can also make effective use of profiling information. Chang, Mahlke, and Hwu [14] use profile data to optimize the most frequently executed portions of a procedure. They build "superblocks," which are chains of basic blocks like those generated by our reordering algorithm, and they duplicate blocks to reduce the number of join points in an attempt to expose more opportunities for optimization. The duplication of blocks is controlled by profile information so that code growth can be bounded by a small constant. Classic optimizations such as constant folding, copy propagation, common subexpression elimination, and dead code elimination are performed at the superblock level.

Chen *et al.* [15] use profile data to guide instruction scheduling by removing constraints that arise from infrequently executed portions of a procedure. McFarling [33] uses profile data and specific information about the target architecture's instruction cache to guide procedure inlining.

# Chapter 5

# Conclusions

This dissertation describes a fast linear intraprocedural register allocation strategy and an efficient infrastructure for profile-driven dynamic recompilation in Scheme. Based on lazy saves, eager restores, and greedy shuffling, the register allocation strategy 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. 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, 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 is remarkably close to optimal in eliminating the need for register shuffling at run time.

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. The 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 [44, 17]. 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.

Other researchers have investigated the use of lambda lifting to increase the number of arguments available for placement in registers [39, 20]. 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 [39] can indeed increase the effectiveness of our register allocator without significantly increasing compile time.

The infrastructure for profile-driven dynamic recompilation enables completely transparent recompilation of procedures, even while they are executing. As a proof of concept, edge-count profile data is used to reorder basic blocks in an attempt to reduce the number of mispredicted branches and instruction cache misses. Performance results show that the log-linear block reordering algorithm and recompilation process are fast, requiring an average of just 12% of the base compile time. In addition, our reordering algorithm effectively reduces the average mispredicted branch rate to a nearly optimal level (7%/6% when static prediction is used) while also decreasing the number of inter-block jumps. Although dynamic branch prediction hardware eliminates some of the need for accurate static prediction, the reordered code still requires an average of 6% less CPU time to execute.

The profile data is obtained using a low-overhead edge-count profiling strategy that supports first-class continuations and reinstrumentation of active procedures, and the profile data can be graphically displayed in terms of the original source. The profiling strategy minimizes compile-time overhead with an efficient log-linear counter placement algorithm. It also employs a fast and effective linear static edge-count estimator that accurately predicts common run-time checks.

Performance results demonstrate that the estimator reduces the average initial run-time profiling overhead from 77% to 50% while very slightly increasing the average compile-time overhead from 10% to 11%. Optimal counter placement based on data from a previous run further reduces the average run-time overhead to 37%, and the reinstrumentation/recompilation process costs 15% of the base compile time. Combined with the block reordering algorithm, the static estimator reduces the average number of statically mispredicted branches to 15%, slightly reduces average run time by 2%, and increases average compile time by just 4%.

The mechanisms described in this dissertation have all been implemented and incorporated into *Chez Scheme*. The recompiler can profile and regenerate all code in the system, including itself, as it runs. One area of future work is to explore various heuristics for deciding which procedures would benefit from recompilation so that profiling and recompilation can be done automatically and transparently as a program executes.

Although Scheme is used to illustrate the details of this research, the techniques and algorithms are not limited to Scheme. The concepts of lazy saves and eager restores should apply to most languages, but the benefits will vary depending on dynamic call behavior. The edge-count profiling strategy should apply to any language. One area of future work is to improve the static estimator's loop prediction strategy without sacrificing linearity. Dynamic recompilation can be done in any language whose implementation can locate all pointers to a given procedure at run time; thus, it is equally well suited to ML, Java, and Smalltalk, among other languages. Moreover, it can be implemented in other languages by providing one level of indirection to procedure calls and returns, a mechanism commonly used by dynamic linkers.

Dynamic recompilation need not be limited to low-level optimizations such as block reordering. A promising area of future work is to associate the profile data with earlier passes of the compiler so that higher-level optimizations such as register allocation, lambda lifting, and procedure inlining can take advantage of the information. For example, our register allocator could assign registers to the most frequently referenced user variables and compiler temporaries rather than assigning them on a first-come, first-served basis. Because these kinds of optimizations destroy the one-to-one correspondence among basic blocks, the collector must be sensitive to return points in original procedures that have no corresponding, compatible return points in recompiled procedures. By retaining original procedures as long as they have live unassociated return addresses, the collector can support these optimizations—even on running procedures.

Another area of future work involves a generalization of edge-count profiling to measure other dynamic program characteristics such as the number of procedure calls, variable references, and so forth. For example, profile data can be used to measure how many times a procedure is called versus how many times it is created, and this ratio could be used to guide lambda lifting [20]. Combined with an estimate of the cost of generating code for the procedure, this ratio could also help determine when run-time code generation [30, 31] would be profitable. Our system could be extended to recompile (specialize) a procedure based on the run-time values of its free variables.

# Bibliography

[1] Kenneth R. Anderson. Courage in profiles. The Fourth International Lisp Users and Vendors Conference, Performing LISP Tutorial, July 1994.

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

[3] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 140–149, June 1994.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[5] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

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

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

[8] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, May 1996.

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

[10] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, May 1996.

[11] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, June 1995.

[12] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme Version 5 System Manual, Revision 2.5*, October 1994.

[13] 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.

[14] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Using profile information to assist classic compiler code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[15] William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Sadun Anik, and Wen-mei W. Hwu. Profile-assisted instruction scheduling. *International Journal of Parallel Programming*, 22(2):151–181, April 1994.

[16] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[17] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, June 1988.

[18] 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.

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

[20] 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.

[21] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically typed languages. Technical Report 400, Indiana University, Computer Science Department, March 1994.

[22] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, 1990.

[23] R. Kent Dybvig and Robert Hieb. An optimistic strategy for argument register saving in Scheme. Indiana University Computer Science Department, April 1991.

[24] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[25] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

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

[27] R. J. Hall. Call Path Profiling. In *International Conference on Software Engineering 14*, pages 296–306, May 1992.

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

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

[30] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, 1994.

[31] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[32] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[33] Scott McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 71–79, June 1991.

[34] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 67–78, June 1995.

[35] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[36] A. Dain Samples. Profile-driven compilation. Technical Report 627, University of California, Berkeley, 1991.

[37] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–366, January 1995.

[38] Robert Sedgewick. *Algorithms*, chapter 31. Addison-Wesley Publishing Company, second edition, 1988.

[39] 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.

[40] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[41] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science Department, Cambridge, Massachusetts, 1988.

[42] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Digital Equipment Corporations's Western Research Laboratory, March 1994.

[43] P. A. Steenkiste. Tags and run-time type checking. In Peter Lee, editor, *Advanced Language Implementation*, chapter 1, pages 3–24. The MIT Press, 1991.

[44] 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.

[45] M. Esen Tuna, Steven D. Johnson, and Robert G. Burger. Continuations in hardware-software codesign. In *Proceedings of the IEEE International Conference on Computer Design*, pages 264–269, October 1994.

[46] Oscar Waddell. SWL reference manual. Indiana University and Cadence Research Systems, August 1996.

[47] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, June 1991.

[48] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, pages 1–42. Springer Verlag, September 1992.

[49] Paul R. Wilson and Thoman G. Moher. Design of the opportunistic garbage collector. In *ACM OOPSLA '89 Conference Proceedings*, pages 23–35, October 1989.

[50] 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.

# Curriculum Vitae

Robert G. Burger graduated as valedictorian from St. John's Lutheran School in LaPorte, Indiana in 1983 and as co-valedictorian from LaPorte High School in 1987.

From 1987 to 1991 he was an Eisenhower Scholar at Rose-Hulman Institute of Technology, where he graduated at the top of his class with a Bachelor of Science degree in the double-major of Computer Science and Mathematics plus a Technical Translator's Certificate in German.

From 1991 to 1992 he maintained large IBM MVS/ESA mainframes and mid-size IBM AS/400 systems as an associate systems engineer at Bristol-Myers Squibb in Evansville, Indiana.

He began graduate studies in Computer Science at Indiana University in 1992 after receiving a three-year Graduate Research Fellowship from the National Science Foundation. In 1994 he completed a Master of Science degree. While working on his doctorate, he was an associate instructor for a sequence of two upper-level compiler design and implementation courses, a research assistant on a National Science Foundation Educational Infrastructure grant, and an independent contractor for Cadence Research Systems. He published several papers [9, 10, 11, 45], and in 1997 he completed a Doctor of Philosophy degree in Computer Science.