Quality and Speed in Linear-Scan Register Allocation

A Thesis presented

by

Omri Traub

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 6, 1998

Table of Contents

Chapter 1 Introduction

Fast compilation tools are essential for high software productivity. Despite the increasing speeds of modern processors, it has never been more important to find and use efficient compilation techniques. As processors become more complex, the demand for highly optimizing code generation is increasing. One response is the trend towards whole-program optimization, where optimizations are targeted towards program units larger than a single procedure or even a single file [7,19]. The success of this approach depends heavily on near-linear optimization techniques. Another growing trend seeks to optimize application code at load or run time. For example, Hoeltzle et al. [12] and Poletto et al. [16] describe the benefits of techniques in adaptive optimization and dynamic code generation respectively. To be acceptably responsive, these techniques must operate at a high speed, so as to avoid incurring a high run time overhead.

The register allocation phase of code generation is often a bottleneck, and yet good register allocation is necessary for making today's processors reach their peak efficiency. It is thus important to understand the trade-off between the speed of register allocation and the quality of the resulting code. In this thesis, we investigate a fast approach to register allocation, called *linear scan*, and compare it to the widely-used graph-coloring method. This comparison shows linear scan to be significantly faster than coloring under most conditions, especially on programs with a large number of variables competing for the same registers. We describe a new linear-scan algorithm, called *second-chance binpacking*, which maintains the linear character but pays much more attention to the quality of the resulting code. As this thesis shows, second-chance binpacking produces code of quality almost identical to that of graph coloring. It is therefore a viable, cost-effective solution.

1.1 The Register Allocation Problem

In order to run efficiently, today's microprocessors must carefully take advantage of the registers which reside on the chip. These are the closest storage units to the CPU's pipeline and therefore have the fastest access time. In a minimal compilation sequence, the front end parses the source high-level language code, performing lexical and semantic analyses, and converting the program into an intermediate representation (IR). The compiler back-end now takes over, translating the intermediate representation into instructions in the target machine's native instruction set. After this phase of code generation, register allocation is performed. At this point, the instructions contain references to three kinds of operands: the program's symbolic variables, compiler-generated temporaries, and machine registers that come pre-assigned due to architectural conventions. Since accessing the registers is much faster than accessing memory (even if the request hits in the first or second level caches), the object of the register allocation phase is to decide which variable and compiler temporaries are *assigned* to registers and which are *spilled* to memory. We seek such a register allocation that minimizes spilling — the traffic between the registers and memory.

We distinguish between *local* and *global* register allocation. Local register allocation seeks to find an assignment of variables to registers within a single basic block — a linear sequence of instructions that are always executed together without interruption. Global register allocation seeks to find an assignment of variables to registers over a procedure's entire control flow graph — a data structure made up of basic blocks, showing the conditional and looping structure of the program code. This thesis considers techniques for global register allocation.

Register allocation has been shown to be *NP*-complete [13,15]. The binpacking formulation of register allocation is in fact equivalent to the 0-1 knapsack problem. Both the graph coloring and the 0-1 knapsack problems are classical *NP*-complete problems [8]. Within these two formulations, researchers have suggested heuristics in an attempt to reduce the complexity of the algorithms. Section 2.7 describes the linear-scan heuristics used in this work.

1.2 Graph-Coloring Register Allocation

In order to find an assignment of register candidates to machine registers, graph-coloring register allocators use information about *liveness* and *interference*. A variable is said to be live at a program point if there is a path to the exit along which its value may be used before it is redefined [15]. It is *dead* if there is no such path. A straight-forward dataflow analysis pass can be used to compute liveness information. Roughly speaking, two variables are said to interfere if they are simultaneously live at some program point.

A graph-coloring allocator summarizes the liveness information relevant to the register allocation problem in an interference graph, where nodes represent register candidates and edges connect two nodes whose corresponding candidates interfere. For a *k*-register target machine, finding a *k*-coloring of the interference graph is equivalent to assigning the candidates to registers without conflict.

The standard graph-coloring method, adapted for register allocation by Chaitin et al. [4,5], iteratively builds an interference graph and heuristically attempts to color it. If the heuristic succeeds, the coloring results in a register assignment. If it fails, some register candidates are spilled to memory, spill code is inserted for their occurrences, and the whole process repeats. In practice, the cost of the graph-coloring approach is dominated by the construction of successive graphs, which is potentially quadratic in the number of register candidates. Since a single compilation unit may have thousands of candidates (variables and compiler-generated temporaries), coloring can be expensive.

1.3 Linear-scan Register Allocation

In contrast to graph coloring, a linear-scan allocator begins with a view of liveness as *lifetime intervals*. A lifetime interval of a register candidate is the segment of the program that starts where the candidate is first live in the static linear order of the code and ends where it is last live. Section 2.1 describes the linear ordering used by linear-scan register allocators. A lifetime typically stretches between a candidate’s first definition and its last use. An exception is looping control flow, where a lifetime interval is extended to loop boundaries where the candidate is live.

A linear-scan allocator visits each lifetime interval in turn, according to its occurrence in the static linear code order, and considers how many intervals are currently active. The number of active intervals represents the competition for available machine registers at this point in the program. When there are too many active lifetimes to fit, a simple heuristic chooses which of them to spill to memory and the scan proceeds. Because it only tries to detect and resolve conflicts locally, rather than for an entire compilation unit at once, linear scan can operate faster than graph coloring. Previous linear-scan allocators run in time linear in the size of the procedure being compiled.

1.4 Our Algorithm

In Chapter 2, we describe our version of a linear-scan allocator. Our algorithm is based on a variant of linear scan, called *binpacking*, that Digital Equipment Corporation uses in its commercial compiler products [1]. We describe several improvements to the binpacking approach. The most significant change involves our algorithm's ability to allocate registers and rewrite the instruction stream in a single scan; all current linear-scan algorithms of which we are aware allocate and rewrite in separate passes. Our algorithm introduces additional flexibility to the register allocation process by giving spilled allocation candidates multiple chances to reside in a register during their lifetimes. Because of this flexibility, our approach requires a second pass to reconcile the linear ordering assumptions with the non-linearity of a procedure's control-flow graph (CFG). In Chapter 3, we describe our experiments. We use the Machine SUIF code-generation framework to compare the performance of our linear-scan algorithm against that of a modern graph-coloring algorithm [9].

Chapter 2 Second-Chance Binpacking

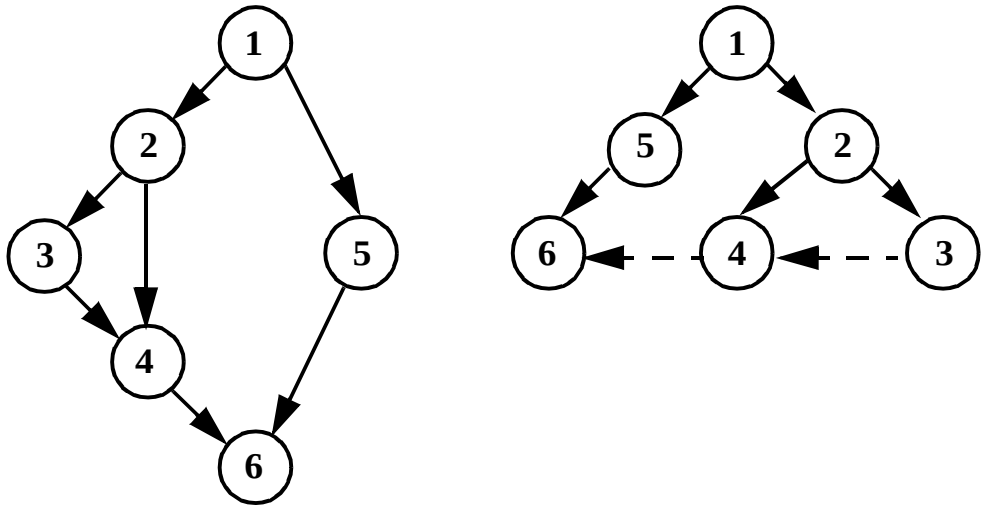
This chapter describes the major contribution of this thesis: a new register allocation algorithm which we call *second-chance binpacking*. Two important goals guide the design of our algorithm: speed of allocation and quality of code produced. In the spirit of the linear-scan family of allocators, we seek to keep the allocation time to a minimum by avoiding expensive, iterative computations such as the ones used in graph-coloring register allocation. Furthermore, unlike any other allocation technique of which we are aware, the algorithm described below performs allocation and code rewriting in a single pass over the instructions of a procedure. In order to meet the goal for quality of produced code, it is clear that we want to minimize the amount of spill code inserted. Section 2.4 and Section 2.6 describe some novel techniques for the minimization of load, store, and move operations. Section 2.5 introduces our methods for resolving the conflicts that remain after the first allocation pass due to the control flow in the program. This resolution phase enables the flexibility of the optimizations in the first pass, where the program is viewed as linear sequence of instructions. We conclude this chapter with an analysis of the complexity of our algorithm and a few observations on the implications of the linear model under which we operate.

2.1 Linear Ordering of a CFG

As its name implies, a linear-scan register allocator needs to fix a linear ordering of the procedure's control flow graph. In a sense, the linear-scan approach abstracts control flow by considering a linear ordering of the basic blocks. The entire control flow graph can then be viewed as a single block.

The linear ordering of choice for most compiler writers is the *reverse postorder*, which is based on a depth-first search of the control flow graph. During a depth-first search, we follow paths in the graph as deeply as possible from a given point before backtracking and trying a different path. More specifically, edges are explored out of the most recently discovered vertex *v* that still has unexplored edges leaving it. When all of *v*'s edges have been explored, the search backtracks to explore edges leaving the vertex from which *v* was discovered [6].

Figure 1 shows an example control flow graph and the tree resulting from the depth-first search.



(a) An example CFG *(b) The corresponding depth-first search tree*

Figure 1. An example of a control flow graph and the linear ordering associated with its blocks.

If we list the order in which nodes are visited in the depth-first search, we get the following sequence:

1, 5, **6**, **5**, 1, 2, **4**, 2, **3**, **2**, **1**

In this sequence, we mark in boldface the last occurrence of each node number. This is the *postorder* of the nodes. The reversal of the postorder is called the *reverse postorder.* Note that the reverse postorder represents an ordering which traverses the graph in the forward direction.

Cormen et al. [6] introduce names for different kinds of edges in the depth-first tree. *Back edges* are defined as those edges (*u*,*v*) connecting a vertex *u* to an ancestor *v* in a depth-first tree. For the purpose of the present discussion, we shall consider all other edges as normal edges. In Figure 1, an edge from block 6 to block 1 representing a loop would have been a back edge in the depth-first tree.

> **Theorem 1:** When numbered in reverse postorder, a node *u* has a higher number than any of its predecessors *v* whenever the edge (*v*,*u*) is not a back edge.
>
> **Proof:** We first observe that the only times a node *u* is listed in the visiting sequence is when it is first discovered or when one of its successors is done and we backup through *u*. Let us now consider the point at which the processing of *u* is complete — i.e. we have finished processing all of its successors. By the observation above, this will be the last time *u* is noted in the visiting sequence. We now backup to the predecessor *p* through which we arrived at *u*. It is clear therefore that *u* will be assigned a higher number than *p*. Consider at this point any other predecessor *v* of *u* such that (*v*,*u*) is not a back edge. There are two cases: *v* may never have been visited, in which case it will be assigned a lower number than *u*; or *v* may have already been visited, in which case it must be an ancestor of *p*, for otherwise we would have discovered *u* through the edge (*v*,*u*). But if *v* is an ancestor of *p*, then *v* will be visited again when backing up from *p*. Thus *v* will be assigned a lower number than *u*. Let us now consider any back edge (*x*, *u*). By definition, *u* is *x*'s ancestor in the tree and would therefore receive a lower number than *x*.

Theorem 1 has important implications for linear-scan register allocators. Since the reverse postorder of the blocks is also the allocation order, we are guaranteed that all of a block's predecessors (except for ones on back edges) have already been allocated when we are about to allocate the block. As we shall see, this property is important when we carry information across block boundaries.

2.2 Allocation Candidates and Lifetime Holes

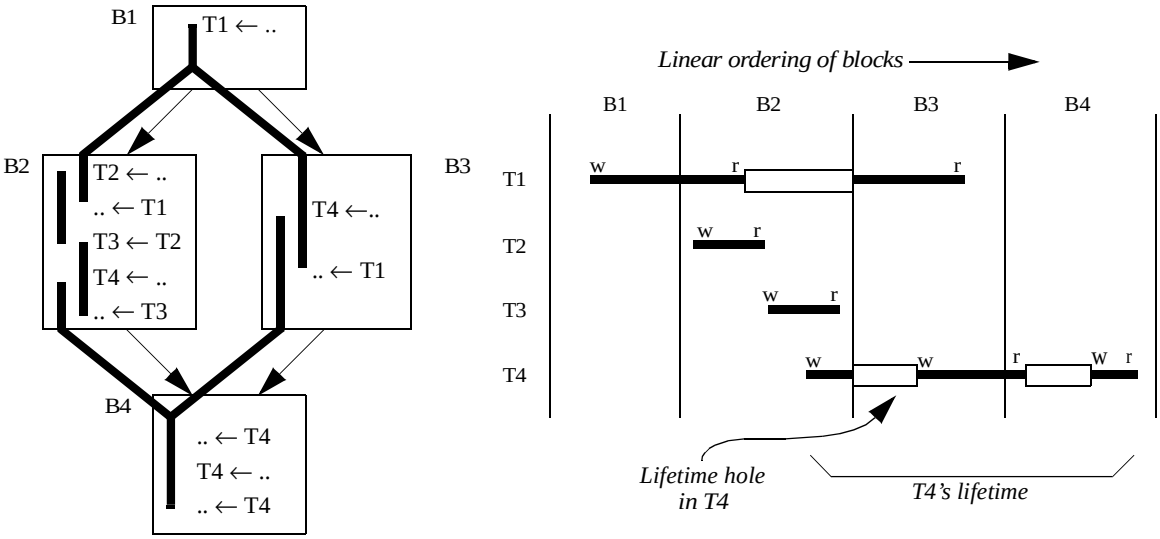
We now describe some preliminary concepts about the objects that we wish to allocate. In our allocator, we seek to assign registers to both program variables and compiler-generated temporaries. We shall refer to all allocation candidates as *temporaries*. We refer to instructions in which a temporary *t* is a source operand as *reading t* and instructions in which *t* is a destination operand as *writing t*.

When examining the lifetime of a temporary, we observe that it may contain one or more intervals during which no useful value is maintained. These intervals are termed *lifetime holes*. Figure 2 illustrates several kinds of lifetime holes that can appear in the lifetime of a temporary. The first kind of hole extends between a use of a temporary and its definition in the same basic block. The last hole in the lifetime of **T4** is of this kind. The second kind starts at a use and extends to the end of a basic block if the temporary is dead on exit from the block and this is the last use in the block. The hole in the lifetime of **T1** is of this kind. The third type of hole starts at the top of a block and extends to the first definition in the block if the temporary is dead on entry to the block. The first hole in the lifetime of **T4** is of such a kind. The last kind, not shown in this figure, occurs when a temporary is dead on entry and exit of a basic block and is never referenced in that block. In that case, a hole extends throughout the block.

Even if we assign a temporary *t* to a register *r* for *t*'s entire lifetime, we can assign another temporary *u* to *r* during *t*'s lifetime if *u*'s lifetime fits inside a lifetime hole in *t*. In Figure 2, temporary **T3** fits entirely in **T1**'s lifetime hole, and thus both could be assigned the same register. We use a single reverse pass over the code to compute lifetimes and lifetime holes.

2.3 The Binpacking Model

The register allocation model that we adopt views the machine registers as bins into which temporary lifetimes are packed. The constraint on a bin is that it may contain only one valid value at any given point in the program execution. Assuming that we had an infinite resource machine with an unbounded number of registers and that our task were to choose the smallest subset of registers that can be assigned to lifetimes, we could minimize this



(a) An example CFG with temporary lifetimes overlaid.

(b) A linear ordering for the example CFG with lifetime holes indicated for each temporary.

Figure 2. Example illustrating the concept of a linear ordering of a procedure's basic blocks, and the lifetimes and lifetime holes for the temporaries in this procedure. Notice that a block boundary can cause a hole to begin or end in the linear view of the program.

number in two ways. First, we could assign two non-overlapping lifetimes to the same register. Second, we could assign two temporaries to the same register if the lifetime of one were entirely contained in a lifetime hole of the other. In each of these cases, the constraint on a register (bin) is never violated.

A binpacking allocator scans the code in a forward linear order, processing the temporaries as they are encountered in the program text. The processing of a temporary *t* involves the allocation of *t* to a register if *t* is not currently assigned a register. We can view an unoccupied register as containing a lifetime hole that extends to a later point in the program where it is no longer free. With this view, the selection of a register to allocate to *t* involves the search for a register with a hole big enough to contain the entire lifetime of *t*. Once we assign *t* to a register *r*, we replace all references to *t* with references to *r* (assuming an infinite number of registers).

In reality, the number of registers available on a given machine is fixed. If at some point in the linear scan there are more overlapping lifetimes than there are available registers, some of these values will need to be spilled into memory. The traditional approach to

linear-scan allocation first traverses the sorted list of lifetime intervals deciding which temporaries should live in a register and which should live in memory. A second phase then scans the procedure code and rewrites each temporary operand with a reference to the appropriate register or to memory. For the purpose of discussion, we assume a load/store architecture where a register is always required. A reference to a spilled temporary is therefore modeled as a point lifetime interval corresponding to a load immediately followed by a use, or a definition immediately followed by a store. These point lifetimes are always assigned a register during allocation.

2.4 Second-Chance Allocation

Early on in the design of our binpacking register allocator, we noticed that it is possible to allocate registers to temporaries and rewrite temporary references all in a single linear pass over the program text. When we encounter a temporary *t* for the first time, we interrupt the rewriting process and determine an allocation for *t*. If we must spill another temporary to create a free register for *t*, we proceed in a manner identical to the approaches that separate the allocation and rewriting phases—a temporary *u* currently residing in a register *r* is spilled to memory and *t* is assigned to *r*. Such spilling decisions are based on a priority heuristic that compares the distance to each temporary's next reference, weighted by the depth of the loop it occurs in, picking the lowest-priority temporary for eviction. Section 2.7 gives the full details on the heuristic function used. Our system is unique among linear-scan allocators in that a spill point marks a split in the lifetime of the evicted temporary *u*. All references to *u* up to this point have already been rewritten to use register *r*. Our algorithm does not go back and change this fact. The spill decision affects only future references to *u*.

When encountering a later reference to this spilled temporary *u*, we must find it a register to occupy during the instruction that uses it. If the reference is a read of *u*, we find a free register *r* (possibly evicting another temporary in the process) and insert a load of *u*'s memory location into *r*. Once we have allocated *u* to this new register *r*, we allow *u* to remain in *r* until some higher-priority temporary evicts it (or *u*'s lifetime ends). In effect, we have split *u*'s lifetime again. The benefit of this approach is that we do not have to reload *u* if we make another reference to it in the near future. We do not need any special

mechanisms to “preference” a later spill load to the same register as the last spill load [3]. In this approach, we optimistically, rather than pessimistically, plan for *u*’s future references. Since we already have to perform lifetime splits due to our emphasis on a single allocate/rewrite pass, our allocator naturally supports this optimistic approach.

If the next reference to a spilled temporary *u* is a write, our allocator uses a similar optimistic heuristic. We allocate *u* to a register *r* (possibly spilling the current temporary in this register), and we postpone the store of this new value for *u* back into memory until some other temporary causes the allocator to evict *u*. All following references to *u* are rewritten to use *r*. If we reach the end of *u*’s lifetime, we may never have to produce the postponed store.

We call our optimistic handling of spilled temporaries *second chance*, because we give temporaries a second (or third, etc.) chance at finding a register home. This second-chance approach is completely generalized to provide a temporary lifetime with a (potentially) new register for every split in its lifetime.

There is one other optimization that we perform while allocating and rewriting. As in the case where we do not create another load of a spilled temporary *t* from memory if *t* is already in a register, we can optimize the rewrite process for adding store instructions. When evicting a temporary *u* from a register *r*, if the value of *u* in *r* matches the value for *u* in memory, we do not add a store of *u*. To perform this optimization, we maintain information about the consistency of the value in *r* with respect to the value in *u*’s memory home. Any load or store of *u* makes the memory home consistent with *r*. Any write of a value to *r* invalidates the consistency of the memory and register values. When we come to a point where we decide to evict *u* from *r*, we avoid the generation of a store spill if *u* is evicted from *r* during one of *u*’s lifetime holes (a store is not needed since the next reference will overwrite the current value) or if the values of *u* in *r* and in memory are consistent.

2.5 Resolution

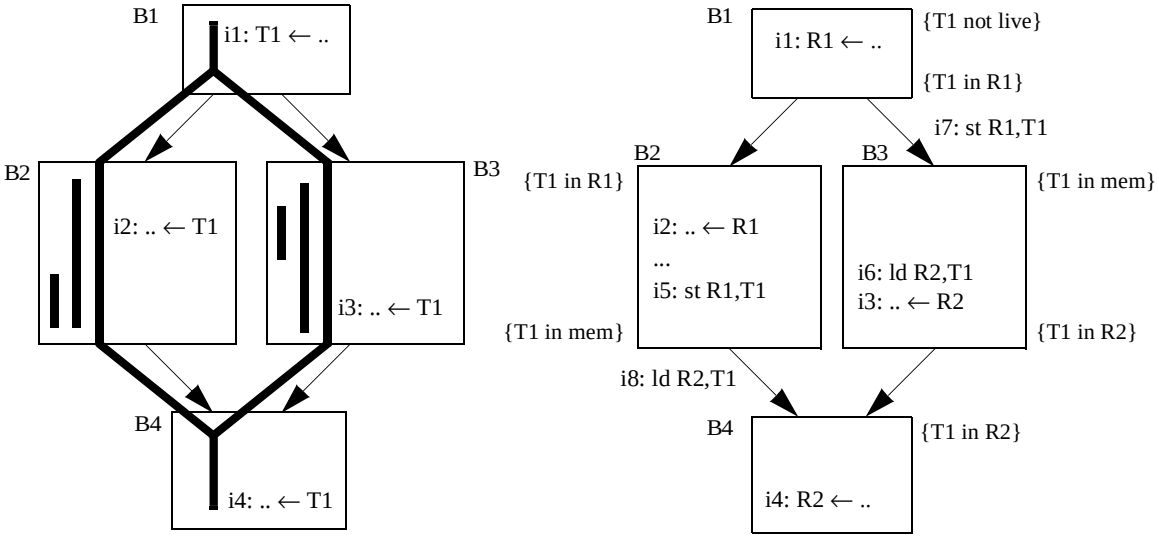
As we mentioned earlier, the more aggressive optimizations of the second chance mechanism come with a cost. In giving a temporary a second chance and assigning it to different

registers at different points in its lifetime, we can potentially create conflicts in the allocation assumptions at basic block boundaries. The linear processing of the allocation/rewrite phase of our approach incompletely models the program control flow. To maintain program semantics, we follow the allocation/rewrite phase with a traversal of the CFG edges, resolving any mismatch in the allocation assumptions across each edge.

We can resolve any conflicts between the allocation assumptions across CFG edges by inserting an appropriate set of load, store, or move instructions. During the allocation pass we maintain a map that gives us information on the location of a temporary at the top and bottom of each basic block. Across a control flow edge, there are three possibilities that require resolution. If the temporary was in a register at the bottom of the predecessor block but in memory at the top of the successor block, we insert[1] a store instruction (but only if a temporary's allocated register and memory home are inconsistent). If the temporary moved from memory to a register, we insert a load instruction. If the temporary was in two different registers across the edge, we insert a move instruction. While processing an edge, we are careful to model the data movement across the edge in a manner that produces the correct resolution instructions in the semantically-correct order, even in the case where two (or more) temporaries swap their allocated registers. This processing is similar to replacing SSA ϕ-nodes by a set of equivalent move operations [14].

Figure 3 gives a simple example of resolution. Assume that none of the temporaries contain lifetime holes and that we have only two registers **R1** and **R2**. The linear allocation order is **B1**-**B2**-**B3**-**B4**. When the allocator encounters **i1** in **B1**, it assigns **T1** to **R1** and rewrites **T1** in **i1** and then **i2** to use **R1**. When the allocator encounters the third lifetime in **B2**, it spills **T1** to memory (**i5**). When it encounters **i3** in **B3**, it inserts a load of **T1** from memory (i6); this time **T1** is given register **R2** — a second-chance allocation. The linear scan completes after rewriting **T1** in **i3** and then **i4** to use **R2**. During resolution, the allocator inserts a store (**i7**) at the top of **B3** and a load (**i8**) at the bottom of **B2**.

1. If the block at the head of the edge has only a single predecessor, we place the resolution code at the top of this block. If the block at the tail of the edge has only a single successor, we place the resolution code at the bottom of this block. If the edge is a critical edge, we split the edge, safely creating a location to place the resolution code.



(a) An example CFG before allocation. The CFG contains 5 temporary lifetimes (thick lines), but only T1's references are shown.

(b) The CFG after allocation. Only instructions associated with T1 *are shown. The allocation assumptions for* T1 *before resolution are shown as sets at the top and bottom of each block.*

Figure 3. Example of conflict resolution at CFG edges.

The linear processing of the CFG can also lead to unnecessary spill loads. Continuing with the example in Figure 3, assume that we remove the shortest lifetime from block **B3**. With this change, the allocator as currently described would still insert the load of **T1** into **R2** for the reference in **i3**. This is because the linear ordering assumes that **T1** is left in memory at the bottom of block **B2** and is therefore also in memory at the top of block **B3**. This is a pessimistic assumption since there is no control-flow edge directly connecting **B2** and **B3**. We would like to be able to take advantage of the fact that one of our registers will be unused from the top of **B3** till **i3** and thus allocate **T1** to this register for the entire length of **B3**. The best choice is to allocate **T1** to **R1** at the top of **B3** (eliminating the generation of any resolution code across the edge **B1-B3**); however, this choice would require us to reconstruct the binpacking state when the linear traversal transitions between two blocks not connected by a control-flow edge. We consider this too expensive an operation considering that **R1** may be needed for another temporary (as in the original example in Figure 3) before the use of **T1** in **i3**. An alternative solution is to run a later code motion pass that tries to sink stores and hoist loads until they meet. When loads and stores to the same memory location meet, we can replace the two operations with a move from the

store's source register to the load's destination register. The resulting move can then be possibly eliminated by subsequent copy propagation and dead-code elimination passes.

Although we do not perform any dataflow analyses during register allocation to minimize the generation or improve the placement of spill code[2], we do perform, during the resolution phase of our allocator, one dataflow analysis for correctness. If we decide not to insert a store instruction when evicting a temporary (see Section 2.4), we assume that the memory and register contents were consistent. This assumption may hold along one or more paths through the control flow graph, but not necessarily through all paths reaching the point where the consistency information is used. In order to determine if and where spill stores need to be inserted to guarantee consistency along all paths, we solve the following iterative bit-vector dataflow problem.

Each bit vector used in our analysis requires as many bits as there are allocation temporaries that are live across basic block boundaries. During the allocation/rewrite phase, we maintain a working bit vector called *ARE_CONSISTENT*. Let A_t be the bit in *ARE_CONSISTENT* corresponding to a temporary *t*. A_t is set as long as *t* is allocated to a register *r* and the contents of *r* are consistent with *t*'s memory home. As described in Section 2.4, a write to *r* clears A_t, and a load or store of *t* set A_t. We will not generate a spill store for *t* during eviction of *t* from *r* if A_t is set. We save a local copy of *ARE_CONSISTENT* at the end of each basic block. This copy is used in the subsequent dataflow analysis.

Also during the first pass, we generate the local GEN and KILL sets for each basic block *b*. The bit vector *WROTE_TR(b)* corresponds to the KILL set. Let W_t be the bit in *WROTE_TR(b)* corresponding to a temporary *t*. W_t is initially clear; it is set whenever a register *r* allocated to *t* is written in *b*. The bit vector *USED_CONSISTENCY(b)* corresponds to the GEN set. Let U_t be the bit in *USED_CONSISTENCY(b)* corresponding to a temporary *t*. U_t is initially clear; it is set whenever W_t is clear and A_t is used to inhibit the

2. We assume the liveness information used in finding lifetimes and holes is available when register allocation begins. The cost of gathering and storing it is amortized over many optimizations in a typical optimizing compiler.

generation of a spill store. In other words, U_t is set whenever the inhibiting of a spill store relies on assumptions of consistency not local to *b*.

Once we have completed the linear scan for the allocate/rewrite phase, we iterate to find a fixed point for the following dataflow equations:

$$USED\_C\_out(b) = \bigcup_{s \in succ(b)} USED\_C\_in(s)$$

$$USED\_C\_in(b) = USED\_CONSISTENCY(b) \cup (USED\_C\_out(b) - WROTE\_TR(b))$$

For all blocks *b*, we initially set *USED_C_in(b)* equal to *USED_CONSISTENCY(b)*.

During resolution processing, we insert a spill store for a temporary *t* during the processing of a CFG edge (*p*,*s*) if the bit for *t* in *USED_C_in(s)* is set and the bit in *ARE_CONSISTENT(p)* is clear. These edges represent the beginnings of paths reaching program points where the consistency of *t*'s register and memory home was exploited, but where the register and memory were not consistent. The placement of this spill store follows the same placement rules as the other resolution code.

2.6 Move Optimizations

Modern architectures typically impose usage conventions for registers. The caller-saved registers, for example, are not preserved across procedure calls. As described so far, our algorithm only allows a temporary to be assigned to a register if that register is free for the temporary's entire remaining lifetime. Under such a restriction, all temporaries live across calls compete solely for the callee-saved registers.

In our algorithm, we represent the constraints on register usage by considering as lifetime holes the intervals during which a register is free. A temporary can now fit inside either a register's lifetime hole or another temporary's lifetime hole. In order to overcome the problem described above, our algorithm allows for a temporary to be assigned to a register with a lifetime hole that is not large enough to contain the entire lifetime. In effect, we are letting the temporary live in the register for as long as the predetermined conventions allow. The algorithm heuristically searches for the largest of these insufficiently-

large holes, trying to leave the temporary in a register for as long as possible. When a register's lifetime hole expires, we check to see if there is still a temporary contained in it. If there is one, we evict the temporary from that register at this point (corresponding to a call site, for example).

When evicting a temporary t from a register r_t that is needed by some architectural convention, we could insert a spill store, reloading t's value the next time we need it through our second-chance mechanism. But it might be true that some other register r_s now has a hole that could contain t's remaining lifetime. If t's lifetime fits in the lifetime hole in r_s, it is more efficient to insert a move from r_t to r_s now than insert a store now and a load later, provided that t is not evicted from r_s before t's next reference. We insert the move at the current point only if we can find an empty register r_s and if a spill store would have been necessary had we not inserted the move. We refer to this mechanism as *early second chance*.

Although a move instruction is more efficient than a load-store instruction pair, we also want to eliminate moves during register allocation when possible. During our linear scan, we perform a check, in the spirit of move coalescing, that attempts to assign both the source and destination of a move to the same register; such moves are eliminated by a separate peep hole optimization pass. Once we assign a register to the source of a move instruction, we check to see if that register has a hole starting immediately after the move's source use and if the lifetime of the move's destination temporary fits within this hole. If so, we bypass the normal allocation mechanism and rewrite the move destination to use the same register as the move source.

Our current implementation performs the move optimization only when the source of a move was already given a register before the allocation phase. This optimization is important since in order to satisfy the Digital Alpha calling convention, our Alpha code generator inserts move operations from the parameter registers to the symbolic names of the parameters at the top of a procedure. We can easily eliminate these moves using our move optimization. If we leave these instructions in the code, they can noticeably degrade the performance of call-intensive programs. It would be straightforward to extend our implementation to attempt move optimization after allocation of a general move source.

2.7 Heuristics

A linear-scan register allocator invokes a heuristic function to decide which of the currently live temporaries have the highest priorities and therefore should belong to a register. In our algorithm, whenever we consider a temporary for a register (either at the beginning of the lifetime or through the second-chance mechanism), we invoke the heuristic to decide which temporary, if any, is to be evicted from a register in order to make room for the new lifetime. In most cases, the new temporary must receive a register for its reference in the current instruction. There are two cases in which a temporary could actually be denied a register. If we are at a loop entry point, a temporary lifetime may extend to the top of the loop when it is live around the back edge. We would like to be able to assign the temporary to a register at the loop top, but there is no actual reference that requires a register. The other case is when a temporary is considered for a register through the early second chance mechanism we described in Section 2.6. We would like to be able to insert a move of the temporary from the register that is now needed for some convention to another free register. In that way, we avoid a store and a future load. It is not absolutely necessary, however, to find a register for the temporary at this point.

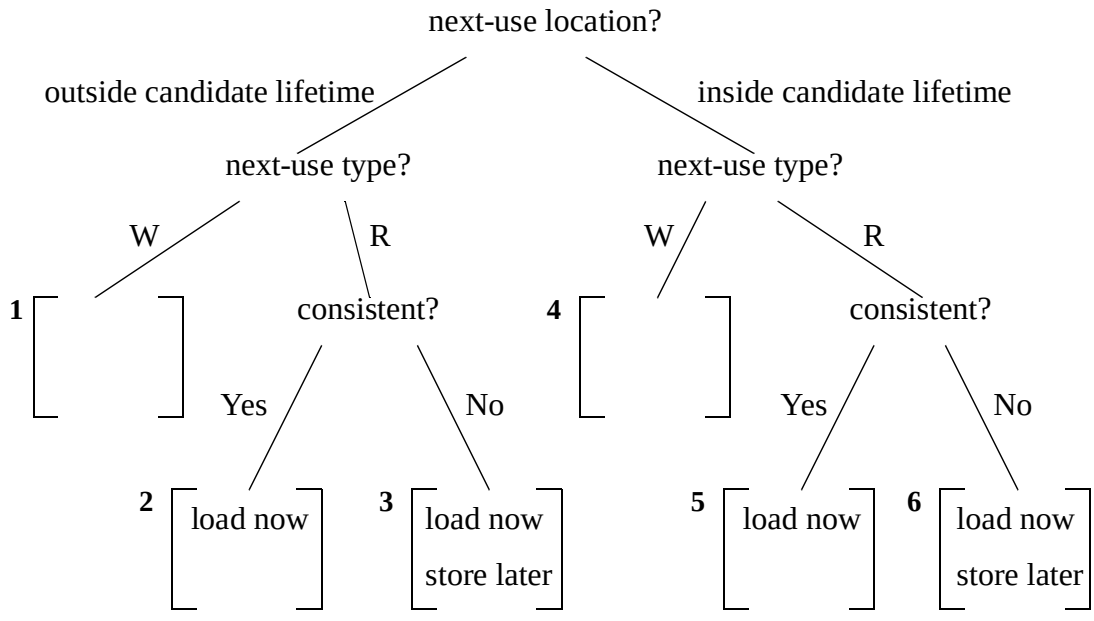


Figure 4. A decision tree representing the various possible heuristic choices in comparing the priorities of two temporaries. All references to next-use and consistency are with respect to the temporary considered for eviction. "Candidate lifetime" refers to the current temporary considered for a register.

Figure 4 shows the decision tree used for the heuristic function. There are six possible outcomes, represented by the leaves of the tree, under three selection criteria. The best case, labeled **1**, occurs when there is either a free register or a temporary with a hole big enough to contain the current candidate lifetime. The empty bracket refers to the null cost of this decision — we need not evict any temporaries to facilitate this allocation. A similar argument holds for choice number **4**. Although the next use of the eviction candidate is within the current candidate's lifetime (the candidate does not fit in the hole), we need not insert any extra instructions for eviction. Because the temporary has a current hole, we need not insert a store: no legal value is held in the register. Since the next reference to the temporary is a write, we will not need a load either. Our heuristic looks for the smallest hole of type **1** in order to leave larger holes available for longer lifetimes. If such a hole is not available, our algorithm searches for the largest hole of type **4** in order to postpone the evicted candidate's next reference for as long as possible.

Although it is clear that the two choices described above are superior to the rest, the relative ordering of the remaining four choices is not so clear. All four possibilities entail additional spill instructions. Ultimately, the decision depends on where these instructions are added. For example, we may prefer not to have to add a store now if we are inside a deeply nested loop.

In creating a priority function, we turn to the optimal solution to the local register allocation problem for some hints. The problem of deciding which of a set of temporaries to evict from a register is akin to the problem in the operating systems domain of choosing which page to evict from memory [18]. The provably optimal solution is to choose the page that is needed furthest in the future, postponing the page fault for as long as possible. When applied to register allocation, this heuristic is called *furthest first*. Since our linear ordering tries to view the program as one basic block, this local heuristic is appropriate. We employ it with a few modifications, using loop depth to assess control flow. The actual formula used is:

$$\frac{loop\_depth\_now \cdot (not\_consistent) + loop\_depth\_at\_next\_use}{distance\_to\_next\_use}$$

The variable *not_consistent* in the formula has the value of 1 if the register and memory values for this temporary are not consistent, and 0 otherwise. The first term represents the cost of a store now if the temporary's value in the register is not consistent with memory. The second term represents the cost of a load in the future since the next reference is a read. We divide the sum by the distance to the next reference to implement the furthest-first heuristic.

Our experience with these heuristics has been that while small changes in the formula may have a relatively large impact on the code, it is very difficult to settle on one heuristic that will be best in all cases. We therefore chose to keep the heuristic relatively simple. As mentioned earlier, code motion optimizations run after allocation may be successful in minimizing and better-placing the spill code.

2.8 Complexity Analysis

We now examine the complexity of our algorithm. We show that it is effectively linear. The running time of the conflict resolution phase is dominated by the dataflow analysis described in Section 2.5 which is not asymptotically linear. As we describe below, however, this dataflow analysis can be replaced by a simpler calculation that does complete in linear time.

The first two phases of the algorithm, computation of lifetimes and holes, then allocation and rewriting, are manifestly linear. Each is a single sweep over the instructions of the program being compiled. Allocation has a constant factor proportional to the number of available registers since it may scan the register state in order to choose and assign a register.

The sweep over edges during conflict resolution is also effectively linear: in real programs most flow nodes have an out-degree of one or two so that the number of edges grows as the number of nodes, and not quadratically.

If the equations for *USED_C_in(b)* and *USED_C_out(b)* given in Section 2.5 are solved by the standard iterative bit-vector calculation, then conflict resolution has a worst-case running time of $O(N^2)$ bit-vector operations, where N is the size of the program. If the size of the bit vectors is the number of temporaries, then the bound is cubic, since the total

number of register candidates is typically proportional to the size of the program. However, the common experience with the standard method is that it terminates in two or three iterations at most, which brings its cost down to *O(N)* bit-vector operations.

In our implementation, the time spent in this dataflow calculation rarely reaches one percent of the time consumed by the overall algorithm. We have therefore not attempted to tune this phase. For situations in which strict linearity is necessary, one could easily replace our iterative dataflow calculation with a more conservative solution. To ensure that we avoid a spill store only when legal, we can conservatively initialize the working copy of the *ARE_CONSISTENT* bit vector at the top of each block *b* encountered during the linear scan. We initialize it with the intersection of the saved *ARE_CONSISTENT* bit vectors at the bottom of all *b*'s predecessor blocks. We assume that any predecessor with an uninitialized bit vector clears all bits in the working bit vector. As Theorem 1 shows, the only predecessors not yet processed by the time we start to allocate a given block are the ones that start a back edge in the control flow graph. So the only blocks for which we don't have complete information are the loop entrances, a small minority of the total number of blocks. Therefore in all but a small number of cases, the above solution would in fact yield the correct consistency state at the top of a block.

In our experiments, conflict resolution including dataflow analysis has never consumed more than five percent of the total time for allocation. Sacrificing strict linearity has not had a major impact.

2.9 Exploring the Linear Model

Abstracting control flow as a linear ordering of the basic blocks makes linear-scan allocators run efficiently. If the allocation decisions in each basic block were independent from the decisions in other blocks, then the order in which we processed the blocks would be immaterial. But in fact some information about the register state and consistency is carried beyond basic block boundaries. This section enumerates the possible edges and transitions in the linear ordering and their effect on this information.

The simplest edge (*u*,*v*) followed in the linear ordering occurs when *u* has no other successors and *v* has no other predecessors. The edge (2,3) in Figure 5 is an example.

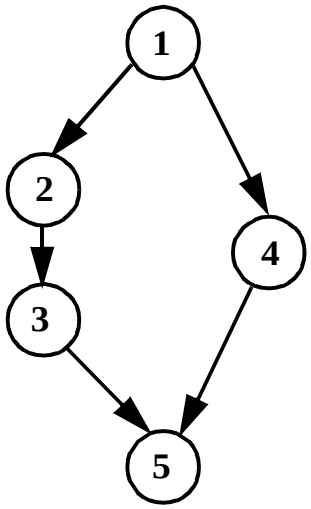


Figure 5. A sample control flow graph.

Since this edge is the only possible transition out of *u* and into *v*, any state existing at the bottom of *u* must also hold at the top of *v*. This kind of edge is relatively rare since the compiler usually collapses the two blocks into a single one.

The next kind of edge occurs when *u* has multiple successors, but *v* has only a single predecessor. This is a *split point* at *u*. The edge (1,2) is an example. We know that any state reaching the bottom of *u* must hold at the top of *v*. Indeed, when following such an edge in the allocation order, we carry the state with us.

The next kind of edge presents more difficulties. This is the case where *u* has a single successor while *v* has multiple predecessors. We say that there is a *join point* at *v*. The edge (4,5) is an example. Now the state leaving *u* certainly reaches *v*, but so does the state from *v*'s other predecessors. This is important when a certain temporary resides in different registers at the bottom of *v*'s different predecessors. As we explain in Section 2.5, the resolution mechanism will fix those edges where the assumptions don't match. In the example above, we carry the register state from block 4 to block 5, leaving it up to resolution to insert code along the edge (3,5) if necessary. But the decision to carry the state from block 4 and not from block 3 is arbitrary. In fact, the edge from 3 may be taken most or all of the time into block 5, as compared with the edge from block 4. We could use profiling information to help us decide which predecessor's state we should take into a given block. But following the philosophy of a linear-scan algorithm, we consider it too expensive to allow

for this choice. If we were to implement such a possibility, we would have to save the register state at the bottom of each block and restore it when we enter a successor. Although possible, this option would introduce considerable complexity since we only maintain information about register contents and their active holes at the current point in the program.

When *u* has multiple successors and *v* has multiple predecessors, the same kind of considerations hold for the edge (*u*,*v*) as for other join points described above.

The final case occurs when we transition from a block *u* to a block *v* without there being a control flow edge between the two blocks. The transition between blocks 3 and 4 in Figure 5 is an example. It does not seem to make sense to carry the register state along this non-existent edge. We do this, however, for two reasons. Again, we believe that it would be expensive to flush and restore the register state. But consider the following example: let T1 be in register R1 at the bottom of block 1, but in register R2 at the bottom of block 3. If we carry the state with us from block 3 to block 4, we would need a resolution move on the edge (1,4). If we avoid this move, however, by restoring the state from the bottom of block 1 when entering block 4 and keeping T1 in R1, we would still need a resolution move into block 5. In summary, it is not clear when it is justified to flush and restore the register state. The simple solution taken seems to fit best the efficiency goals of our allocator.

In Section 2.5 we discussed the consistency information that is also carried between basic blocks. As Section 2.8 points out, it is possible to set the consistency bit vector at the top of a block equal to the intersection of the saved consistency bit vectors at the bottom of each of its predecessors. This would make sense in this situation and would not involve a high overhead since we save this information at the bottom of blocks in any case. In a future implementation we will try to compare this approach against our current dataflow analysis.

Chapter 3 Experimental Evaluation

To compare fairly our linear-scan register allocator with a graph-coloring allocator, we have implemented them both in the Machine SUIF extension [17] of the Stanford SUIF compiler system [20]. SUIF makes it easy to mix and match compiler passes. Keeping the rest of the compiler fixed, we created two alternative register allocation passes, identical in every respect except for the central allocation algorithm. We compare the allocation approaches along two dimensions. We first assess the quality of the code produced by examining the run time of different benchmarks when allocated using each approach. By instrumenting the benchmark executables, we are able to collect dynamic instruction counts for the program run. We also compare the running time of each of the allocators on the benchmark input. This cost is part of the compile time for each program.

In implementing the two approaches, second-chance binpacking and graph coloring, we are careful to use a common framework so that any difference in performance is due strictly to the different algorithms used. In both passes, for example, we use shared libraries to construct CFGs and perform liveness and loop-depth analyses, attaching the results to the CFG prior to register allocation. Moreover, we use a common set of utilities for scanning the code and updating it to insert spill instructions or to reflect register assignments. Loop depth is used in the same way to weight occurrence counts in both allocators.

The coloring method used is an implementation of the algorithm described by George and Appel [9]. This is a pure coloring approach in the style originated by Chaitin [5] and refined by Briggs et al. [2]. Its principal departure from that style is that it integrates register coalescing (copy propagation) into the coloring phase of allocation, rather than performing it repeatedly beforehand in a loop. Register coalescing attempts to unify two

allocation candidates into one when they do not interfere and when a move (copy) instruction connects their lifetimes. The usual Chaitin-Briggs method builds a new interference graph after each successful round of coalescing. George and Appel take the costly graph-building operation out of the inner loop. They report that it also improves code significantly by eliminating more copy instructions. Our implementation is faithful to the published algorithm [9] with two exceptions:

- We use a lower-triangular bit matrix, rather than a hash table, to record the adjacency relation of the interference graph.
- We perform liveness analysis only once, before allocation, rather than once per round of coloring. For both linear scan and graph coloring, temporaries that are live only within a single basic block are excluded from dataflow analysis, which greatly reduces bit vector sizes and makes repeated dataflow analysis unnecessary between coloring iterations.

The latter simplification is possible for both linear scan and graph coloring because the temporaries generated by spill code insertion are live only within a single basic block. Global liveness information is not affected by such temporaries.

When targeting the Digital Alpha, our graph-coloring allocator deals separately with general-purpose registers and floating-point registers. On current Alpha implementations, data moved between register files must go through memory. Each register operand of a given instruction can only reside in one file or the other. With coloring, the non-linear costs of building the interference graph and choosing temporaries to spill make it more efficient to solve the two smaller problems separately. (This is the approach used, for example, in the compiler for which George and Appel designed their algorithm.) Our linear-scan algorithm, on the other hand, processes both register files at once.

3.1 Run Times

We compare the quality of generated code on a number of benchmarks. Most benchmarks are from the SPEC92 suite, except for *compress* and *m88ksim* (SPEC95) and *sort* and *wc* (UNIX utilities). The target machine for these experiments is a Digital Alpha running Digital UNIX 4.0.

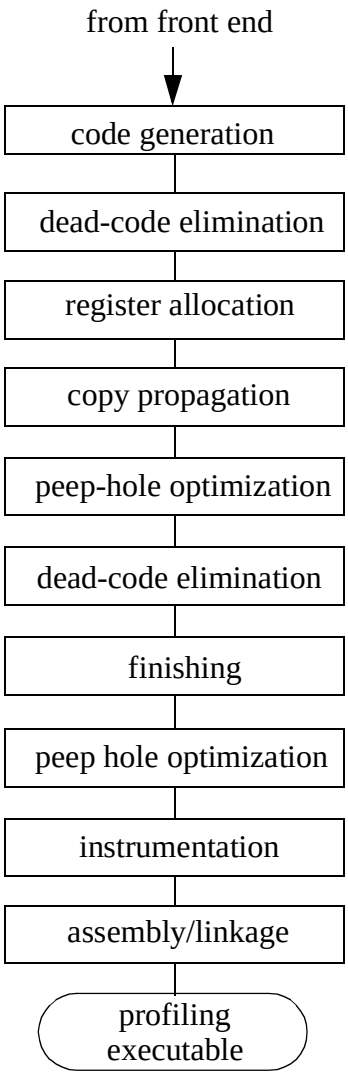


Figure 6. Flow of compilation passes for dynamic instruction counts experiment.

In order to assess the quality of the code produced by each allocation approach, we add instrumentation code to collect the dynamic instruction counts for each benchmark. These results were obtained using the HALT tool within Machine SUIF to instrument each benchmark after code generation. Figure 6 illustrates the sequence of passes used to compile each benchmark. At different points in the compilation, peep hole optimization is applied to remove unnecessary move instructions and collapse pairs of adjacent instructions into one whenever possible. An example would be eliminating a move that follows an add instruction by rewriting the destination register in the add. The coloring implementation performs register coalescing, but the linear-scan approach does not. We therefore

Benchmark	Instruction counts		
	Second-chance binpacking	Graph coloring	Ratio (binpack/GC)
alvinn	5812393811	5812393811	1.000
doduc	1488898698	1486641792	1.002
eqntott	2614155547	2614011368	1.000
espresso	1281934463	1265263844	1.013
fpppp	6527750012	6205402412	1.052
li	9442694155	9274298199	1.018
tomcatv	5770774584	5770748890	1.000
wave5	13268676858	13264335177	1.000
compress	88420629554	88210696298	1.002
m88ksim	1006565812	998834549	1.008
sort	958984834	926740533	1.035
wc	1038259	1038247	1.000

Table 1: A comparison of the dynamic instruction counts of executables using either our second-chance binpacking approach or George/Appel's graph-coloring approach.

run copy propagation after register allocation. Finally, a dead-code elimination pass is added to remove unnecessary instructions and basic blocks from the program.

In addition to dynamic instruction counts, we also report run times in seconds of execution. These results were obtained with the UNIX time() command on a lightly-loaded Alpha. Each time is the best of five consecutive runs.

Table 1 and Table 2 present the run-time results. For each metric, we calculate the ratio of the result under linear scan to the result under graph coloring. Larger ratios mean poorer performance of the linear-scan-produced executable.

Overall, our approach produced executables that are of a quality near to those produced by coloring. In most benchmarks the performance degradation of linear scan over coloring does not exceed one percent. In five out of the twelve benchmarks studied, the two allocation approaches produced code of virtually identical quality. For the *sort* and *fpppp* benchmarks, the excess of dynamically executed instructions under linear scan was 3.5% and 5.2% respectively. These benchmarks contain code with very high register pressure which we observed to be very sensitive to small variations in the heuristic function

Benchmark	Run time (sec) Second-chance binpacking	Graph coloring	Ratio (binpack/GC)
alvinn	20.57	20.57	1.000
doduc	7.02	7.09	0.990
eqntott	6.76	6.75	1.001
espresso	3.28	3.16	1.038
fpppp	25.77	25.51	1.010
li	23.33	23.71	0.984
tomcatv	12.93	13.04	0.992
wave5	40.21	40.30	0.998
compress	269.11	269.56	0.998
m88ksim	2.51	2.49	1.008
sort	4.02	3.84	1.047
wc	0.84	0.84	1.000

Table 2: A comparison of the run times of executables using either our second-chance binpacking approach or George/Appel's graph-coloring approach.

used in the linear-scan algorithm. As we mention in Chapter 2, we believe that later passes that perform code motion optimizations can improve the placing of spill code.

The run time results reported in Table 2 also show very similar performance of the code produced by the different approaches. A number of benchmarks seem to actually run faster when allocated with our linear-scan approach. Although the timing numbers were stable across five runs, we believe that the result of the UNIX time() command is not as good an indicator of performance as the dynamic instruction counts we report.

To help explain the variation in the instruction count results, Table 3 presents information on the percentage of the total dynamic instruction count due to spill code inserted by the register allocator. We count load, store, and move instructions inserted for allocation candidates only. Five of our benchmarks (*alvinn*, *li*, *tomcatv*, *compress*, and *wc*) had no spill code added by either approach. In a sense both allocators did as well as possible for these programs. For these applications, the difference in the dynamic instruction counts in Table 1 is due to differences in spill code placement. We also observed that the coalescing phase in the coloring allocator is able to remove more copy (move) instructions than the copy propagation pass in its classical formulation [15].

Benchmark	Second-chance binpacking	Graph coloring
alvinn	0%	0%
doduc	0.493%	0.518%
eqntott	0.001%	0.000%
espresso	0.901%	0.163%
fpppp	17.111%	13.521%
li	0%	0%
tomcatv	0%	0%
wave5	0.046%	0.032%
compress	0%	0%
m88ksim	0.033%	0.049%
sort	1.438%	0.966%
wc	0%	0%

Table 3: Percentage of total dynamic instructions due to spill code for each allocation approach. If no spill code was inserted during register allocation, the percentage is reported as simply "0%".

For the applications with spill code, Figure 7 presents a detailed look at the composition of the spill code produced both by second-chance binpacking and by graph coloring. For both *doduc* and *m88ksim*, binpacking produced less spill code than coloring. The majority of the difference is due to the insertion of extra spill loads during coloring. Our binpacking allocator produced more spill code than coloring for *eqntott*, *espresso*, *fpppp*, *sort*, and *wave5*. A significant proportion of this increase appears due to extra stores (resolution and eviction). These stores can, as in the case of *eqntott*, lead to a large number of resolution loads. A review of the output code shows that a global optimization pass run after allocation can eliminate unnecessary load/store pairs as well as partially redundant spill instructions using hoisting and sinking techniques.

In order to evaluate the advantages of our second-chance binpacking over traditional two-pass binpacking, we created a version of our allocator that assigns a whole lifetime to either memory or register. This implementation still takes advantage of lifetime holes during allocation. We observed two classes of applications with respect to the performance of this allocator. The first, represented best by the word-count (*wc*) benchmark, contains those applications whose performance degrades substantially under binpacking without second chance. The *wc* benchmark ran 38% slower (1436991 vs. 1038259 dynamic instructions) when allocated using two-pass binpacking than it did when allocated with

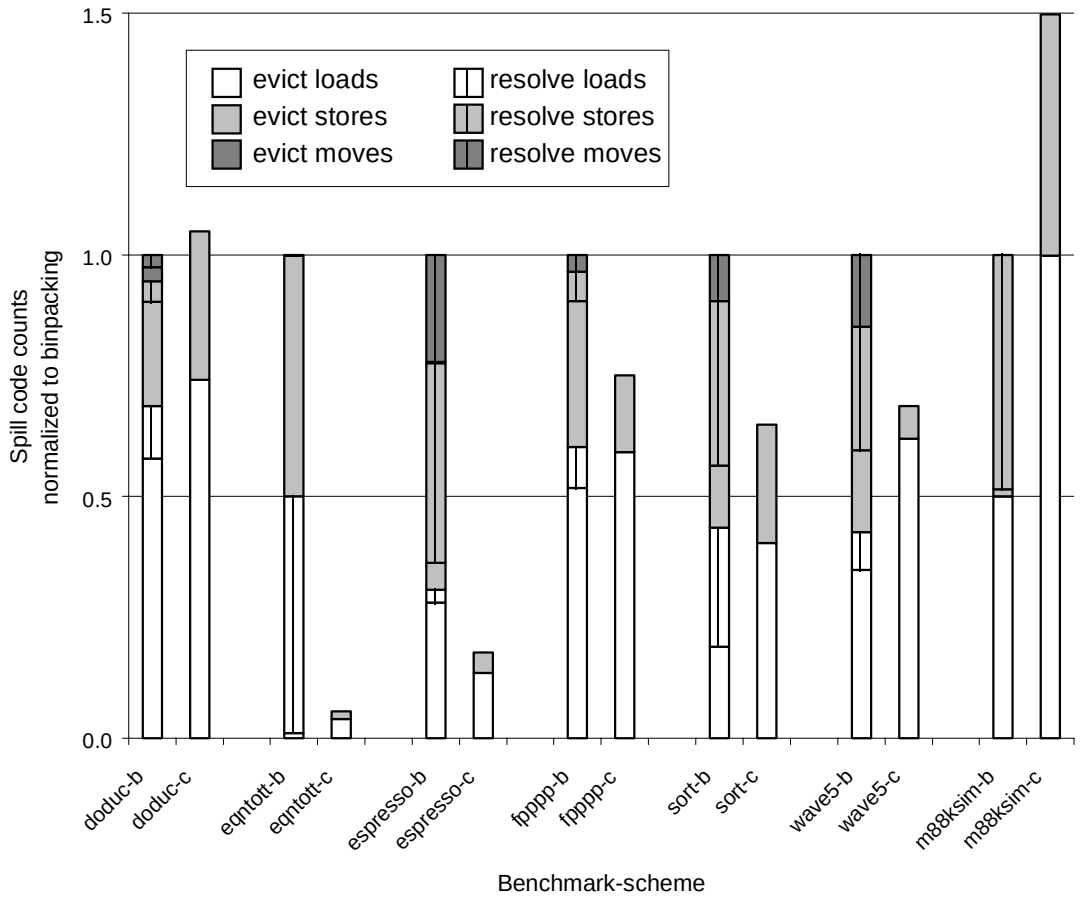


Figure 7. A categorization of the spill code inserted by each allocator. Results for our binpacking approach are labelled with a "-b" while those for coloring are labelled with "-c". For each benchmark, we normalize the counts to the total spill code inserted with binpacking. We have separated the "eviction" spill code inserted during our linear scan and the coloring algorithm's spill phase from the "resolve" spill code inserted during our resolution phase.

our second-chance approach. The *wc* benchmark has a large number of temporaries that are live throughout a loop that contains a procedure call to an I/O routine. Our second-chance mechanism manages to allocate some of the temporaries to caller-saved registers, evicting them just before the procedure call but avoiding unnecessary stores. The two-pass binpacking approach, however, is not able to use the caller-saved registers (there is no hole in a caller-saved register large enough to contain the lifetimes of the temporaries live across the call). It thus evicts temporaries out of the callee-saved registers. Since this algorithm does not avoid unnecessary stores, costly spill code is inserted inside the loop. The other class of applications, exemplified by *eqntott*, has almost identical performance under two-pass binpacking and second-chance binpacking (2615277744 vs. 2614155547

dynamic instructions). The *eqntott* benchmark spends the majority of its time in the procedure cmppt(), which contains a very small number of temporaries and therefore requires no spilling.

3.2 Compile Times

To evaluate the compilation speed of the two methods, we timed both on representative modules from the benchmark set. Table 4 shows results obtained by timing only the core parts of the allocators on a lightly-loaded Alpha. In particular, we record the time of day after setup activities common to both allocators (CFG construction, loop analysis, liveness analysis, etc.), and then record the time of day again after allocation. The difference in these two recorded times is summed over all procedures in a compiled module to produce the times in Table 4. Each is the best of five consecutive runs. The table also includes the average number of register candidates per procedure in the module and the average number of edges in their interference graphs.

Module or Procedure (Benchmark)	Average number of		Allocation time (sec)	
	Register candidates	Interference graph edges	Graph coloring	Second-chance binpacking
cvrin.c (espresso)	245	1061	0.4	1.5
twldrv.f (fpppp)	6218	51796	8.8	3.7
fpppp.f (fpppp)	6697	116926	15.8	4.5
field_() (wave5)	7611	86741	14.9	4.94

Table 4: A comparison of the allocation times.
The average number of register candidates and interference graph edges refer to the coloring allocator. These numbers cover all coloring iterations.

While the coloring allocator is actually faster on small problems, its performance rapidly becomes worse on programs with many competing register candidates. These numbers illustrate that a coloring allocator slows down significantly as the complexity of the interference graph increases.

The data from the *wave5* benchmark is particularly interesting. Despite the much shorter time spent on allocation in the linear-scan algorithm, the actual performance of the resulting code was virtually identical to that of the code allocated with graph coloring (see Table 1 and Table 2).

Chapter 4 Related Work

The phrase “linear scan” was used by the developers of the ‘C dynamic code generator to describe the register allocator in their system [16]. Having tried graph coloring, they developed a simpler method that scans a sorted list of the lifetimes and at each step considers how many lifetimes are currently active and thus in competition for the available registers. When there are too many active lifetimes to fit, the longest active lifetime is spilled to memory and the scan proceeds. No attempt is made to take advantage of lifetime holes or to allocate partial lifetimes. Nevertheless, in the context of a run-time code generator, the improvement in compilation speed obtained by using linear scan instead of coloring justifies a modest decrease in run-time performance.

Digital Equipment Corporation has used a linear-scan algorithm for many years in the GEM optimizing code generator, a compiler back-end used in several of its compiler products [1]. The GEM approach to binpacking and treatment of lifetime holes [3] were the starting points for our work on linear-scan allocation. Binpacking evolved from research done in the production quality compiler-compiler project at CMU [13,21]. However, the discovery of linear-scan register allocation at Digital was almost an accident: its first implementation was intended as a “throw-away” module, meant to be replaced by a more elaborate scheme. When the throw-away turned out to perform better than its more complicated replacement, it was shipped with the product instead [11].

Digital’s allocator uses “history preferencing”, which allows load instructions to be omitted by remembering which values in memory are mirrored in registers. Our second-chance method subsumes history preferencing and adds the dual optimization of avoiding

a store instruction when a register's value can be shown to exist in memory already or never to be needed in memory again.

Laurie Hendren and a group from McGill University have experimented with an alternative representation for interference graphs which they call *cyclic interval graphs* [10]. This data structure provides more fine grain information about the overlap between two temporary lifetimes, especially those extending around a loop. Hendren's algorithm covers points of maximal pressure with a *fat cover*, a set of non-overlapping intervals that can fit into one register. This idea is very similar to binpacking. Hendren also introduces the concept of a *chameleon interval*, a temporary that is assigned different colors, or registers, at different points in its lifetime. This concept is related to the second-chance mechanism that we present, only Hendren restricts her allocator to a single basic block and therefore does not have to perform any resolution as a result of the multiple register homes.

In his recent book, Bob Morgan presents a hybrid approach to register allocation [14]. He first runs a limiting pass which reduces the register pressure by introducing spill code for temporaries that are live through loops. He then runs his register allocator in three phases: he starts by using graph coloring to allocate temporaries that are live across basic blocks. He then uses Hendren's representation and algorithm to allocate those local temporaries that can occupy the same registers as the global temporaries. His final phase uses a standard local algorithm to allocate the purely local temporaries.

Chapter 5 Conclusions

Linear-scan methods of register allocation are fast and effective. They can enable the interprocedural optimization of large programs, and are appropriate for run-time code generation. They avoid the risk of the compile-time performance degradation that graph-coloring methods suffer on certain program inputs.

We have presented and studied a new implementation of linear scan, called second-chance binpacking. This approach performs register allocation and instruction rewriting in a single pass, paying more attention to spill code minimization than other linear-scan approaches. We have made a fair comparison of this new method with a well-designed coloring algorithm and found linear scan to be competitive in output quality and much less prone to slow down on complex inputs. On the benchmarks studied, the performance of code produced by the linear-scan algorithm was never more than 5% worse than the performance of the code produced by the coloring algorithm. On many benchmarks, the performance was identical under the two approaches.

The results of this thesis demonstrate a valuable lesson in algorithm and software design: understanding the trade-offs between the time spent by an algorithm and the quality of the output produced is an important step in the design of efficient systems. The surprising efficacy of the linear-scan approach to register allocation reminds us of the intricate nature of the *NP*-complete problem space.

References

[1] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. P. Grove, S. O. Hobbs and W. B. Noyce, “The GEM Optimizing Compiler System,” *Digital Equipment Corporation Technical Journal*, 4(4):121–135, 1992.

[2] P. Briggs, K. Cooper, and L. Torczon, “Improvements to Graph Coloring Register Allocation,” *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[3] C. K. Burmeister, K. W. Harris, W. B. Noyce and S. O. Hobbs, U.S. patent number 5,339,428.

[4] G. Chaitin et al., “Register Allocation via Coloring,” *Computer Languages*, 6, 47–57, 1981.

[5] G. J. Chaitin, “Register Allocation and Spilling via Graph Coloring,” *ACM SIGPLAN Notices*, 17(6):201–107, June 1982.

[6] T. H. Cormen, C. E. Leiserson, and R. L Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[7] M. F. Fernandez, “Simple and Effective Link-time Optimization of Modula-3 Programs,” *ACM SIGPLAN Notices*, 30(6):103–115, June 1995.

[8] M. R. Garey, and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freedman and Company, New York, NY, 1979.

[9] L. George and A. Appel, “Iterated Register Coalescing,” *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[10] L. J. Hendren, G. R. Gao, E. R. Altman and C. Mukerji, “A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs,” *Proc. 4th International Compiler Construction Conference*, 176–191, October 1992.

[11] S. O. Hobbs, Personal communication, July 1997.

[12] U. Hoeltze, “Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming,” Ph.D. thesis, Stanford University, March 1995.

[13] B. Leverett, “Register Allocation in Optimizing Compilers,” Ph.D. thesis, CMU-CS-81-103, Carnegie-Mellon University, February 1981.

[14] R. Morgan, *Building an Optimizing Compiler*, Digital Press, Boston, MA, 1998.

[15] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[16] M. Poletto, D. R. Engler and M. F. Kaashoek, “tcc: a System for Fast, Flexible and High-level Dynamic Code Generation,” *ACM SIGPLAN Notices*, 32(5):109–121, May 1997.

[17] M. Smith, “Extending SUIF for Machine-dependent Optimizations,” *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 14–25, January 1996. URL: http://www.eecs.harvard.edu/machsuif.

[18] A. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[19] D. W. Wall, “Global Register Allocation at Link Time,” *ACM SIGPLAN Notices*, 21(7):264–275, July 1986.

[20] R. Wilson et al., “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers,” *ACM SIGPLAN Notices*, 29:31–37, May 1994. URL: http://suif.stanford.edu.

[21] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.

Acknowledgments

I would like to first thank my advisor, Prof. Mike Smith, for his continuous support of this project. In all stages of the design and implementation he was always involved, unwilling to compromise on the quality of the research or the experimental results. From him I learned invaluable lessons in compiler design, in software engineering, and even in the intricate art of evaluating and debugging assembly code.

I would also like to thank Glenn Holloway for his incredible efforts and endless patience. I can always count on him for answers to any question: his knowledge of computer science is enormous. Glenn is also the implementer of the graph-coloring algorithm used in the comparison in Chapter 3.

To my committee members, Profs. Harry Lewis and Margo Seltzer, I wish to extend many thanks for taking the time to read this thesis. Harry Lewis has taught me important lessons in the theory of *NP*-completeness, algorithmic design, and mathematical thinking.

A number of people outside Harvard have also been supportive of this research. Steve Hobbs and Bob Morgan of Digital Equipment Corporation were very helpful in discussing the implementation of binpacking in the GEM compiler. Max Poletto from MIT was helpful in his explanation of the use of linear scan in dynamic code generation.

Finally, I would like to thank my family and friends, Flora Stern in particular, for their moral support and encouragement during the many months of work on this research.