Quality and Speed in Linear-scan Register Allocation

Omri Traub, Glenn Holloway, Michael D. Smith
Harvard University
Division of Engineering and Applied Sciences
Cambridge, MA 02138
{otraub, holloway, smith}@eecs.harvard.edu

Abstract

A *linear-scan* algorithm directs the global allocation of register candidates to registers based on a simple linear sweep over the program being compiled. This approach to register allocation makes sense for systems, such as those for dynamic compilation, where compilation speed is important. In contrast, most commercial and research optimizing compilers rely on a graph-coloring approach to global register allocation. In this paper, we compare the performance of a linear-scan method against a modern graph-coloring method. We implement both register allocators within the Machine SUIF extension of the Stanford SUIF compiler system. Experimental results show that linear scan is much faster than coloring on benchmarks with large numbers of register candidates. We also describe improvements to the linear-scan approach that do not change its linear character, but allow it to produce code of a quality near to that produced by graph coloring.

Keywords: global register allocation, graph coloring, linear scan, binpacking

1 Introduction

Fast compilation tools are essential for high software productivity. 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 speed of register allocation and the quality of the resulting code. In this paper, we investigate a fast approach to register allocation, called *linear scan*, and we compare it to the widely-used graph-coloring method. This fair comparison shows linear scan to be faster than coloring under most conditions, especially on programs with large numbers of variables competing for the same registers. Since emitting high quality code was our first priority in implementing our linear scan allocator, we describe some novel improvements to the linear-scan approach that improve output code without destroying the linear character of the algorithm.

Despite the increasing speeds of modern processors, it has never been more important to find and use efficient compilation techniques. The demand for highly optimizing code generation is increasing as processors become more complex. One response is the trend towards whole-program optimization [6,15]. 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. [10] and Poletto et al. [13] describe the benefits of techniques in adaptive optimization and dynamic code generation respectively. To be acceptably responsive, these techniques must operate at speeds measured in a reasonable number of cycles per generated instruction.

Both graph-coloring and linear-scan allocators use liveness information to find an assignment of register candidates to the machine registers. To achieve this goal, a graph-coloring allocator summarizes liveness information as an interference graph, with nodes representing register candidates and edges connecting nodes whose corresponding candidates are live at the same time and therefore cannot coexist in a register. 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 (including compiler-generated temporaries), coloring can be expensive.

In contrast to graph coloring, a linear-scan allocator begins with a view of liveness as a *lifetime interval*. 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. 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

Appears in the Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pages 142–151, June 1998.

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.

In Section 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. By allocating and rewriting simultaneously, we introduce flexibility into 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 resolve the linear-scan assumptions with the non-linearity of a procedure's control-flow graph (CFG). Because the second pass entails a dataflow analysis, its worst-case asymptotic complexity is quadratic in the program size. However, as we explain in Section 2.6, it can be engineered to give linear performance in all cases. In Section 3, we describe our experiments, which use the Machine SUIF code generation framework to compare the performance of our linear-scan algorithm against a modern graph coloring algorithm [7]. Section 4 discusses related efforts in linear-scan register allocation, and Section 5 summarizes our contributions.

2 Second-chance binpacking

Two important goals guide the design of our register allocation 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. This approach influences many of our design decisions. After Section 2.1 introduces the general concepts behind a binpacking allocator, Section 2.2 outlines the technique and focuses on the novel aspects of our algorithm. Section 2.3 describes how we handle spills during the linear allocate/rewrite phase, while Section 2.4 discusses the second phase of our algorithm which resolves the assumptions made during the linear first phase with the non-linear flow of a CFG. Section 2.5 presents two optimizations related to the creation of spill code and the elimination of moves. Section 2.6 summarizes the computational complexity of our algorithm.

2.1 Allocation candidates and lifetime holes

We begin by describing 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 generically as *temporaries*.

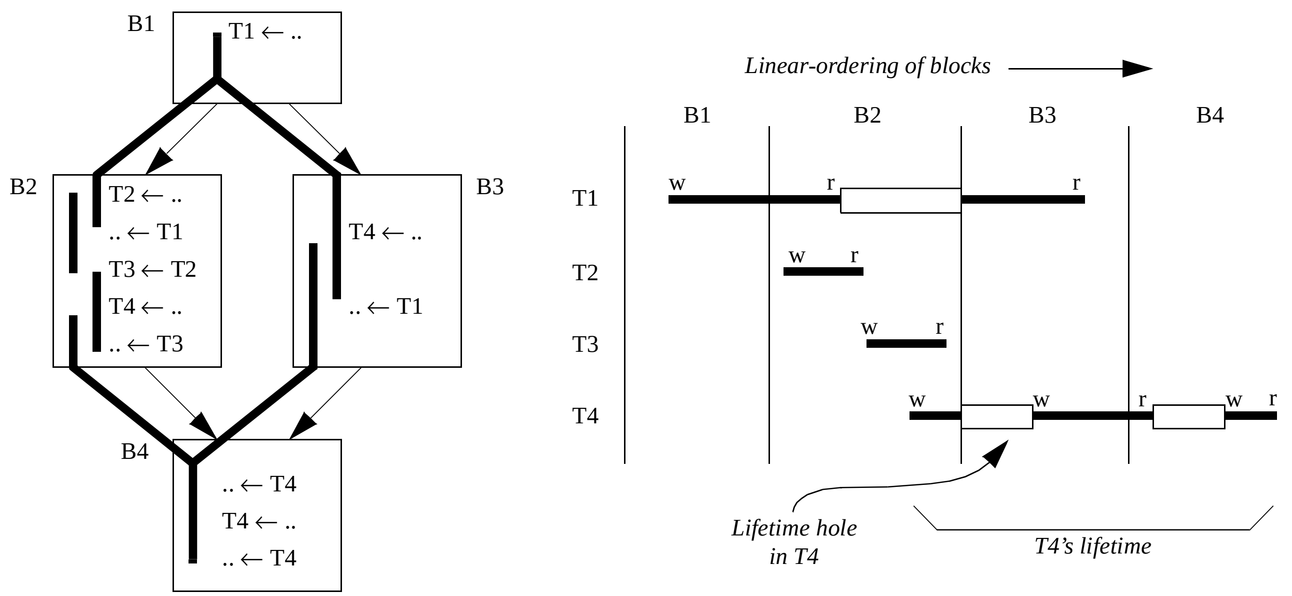
When examining the lifetime of a temporary, we observe that it may contain one or more intervals during which no useful value is maintained. These intervals are termed *lifetime holes*. Figure 1 illustrates several kinds of lifetime holes that can appear in the lifetime of a temporary. Even if we assign a register *r* to a temporary *t* 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 1, 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.

2.2 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 have an infinite resource machine with an unbounded number of registers and that our task is to choose the smallest subset of registers that can be assigned to lifetimes, we can minimize this number in two ways. First, we can assign two non-overlapping lifetimes to the same register. Second, we can assign two temporaries to the same register if the lifetime of one is entirely contained in a lifetime hole of the other. Under both these approaches, the constraint on a register (bin) is preserved.

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*. If we have multiple registers with holes large enough to contain *t*'s entire lifetime, we heuristically choose the register with the smallest hole that is larger than *t*'s lifetime. Once we assign *t* to a register *r*, we would replace all references to *t* with references to *r* (assuming infinite 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 walks the sorted list of lifetime intervals deciding which temporaries live in a register and which live in memory. A second phase then scans the procedure code and rewrites each operand with a reference to the appropriate register or to memory. For the purposes of discussion, we assume a load/store architecture where a register is always required,



(a) An example CFG with temporary lifetimes overlaid.

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

Figure 1. 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.

and so a reference to a spilled temporary is modeled as a point lifetime interval corresponding to the load or store of the spilled temporary. These point lifetimes are always assigned a register during allocation.

2.3 Second-chance allocation

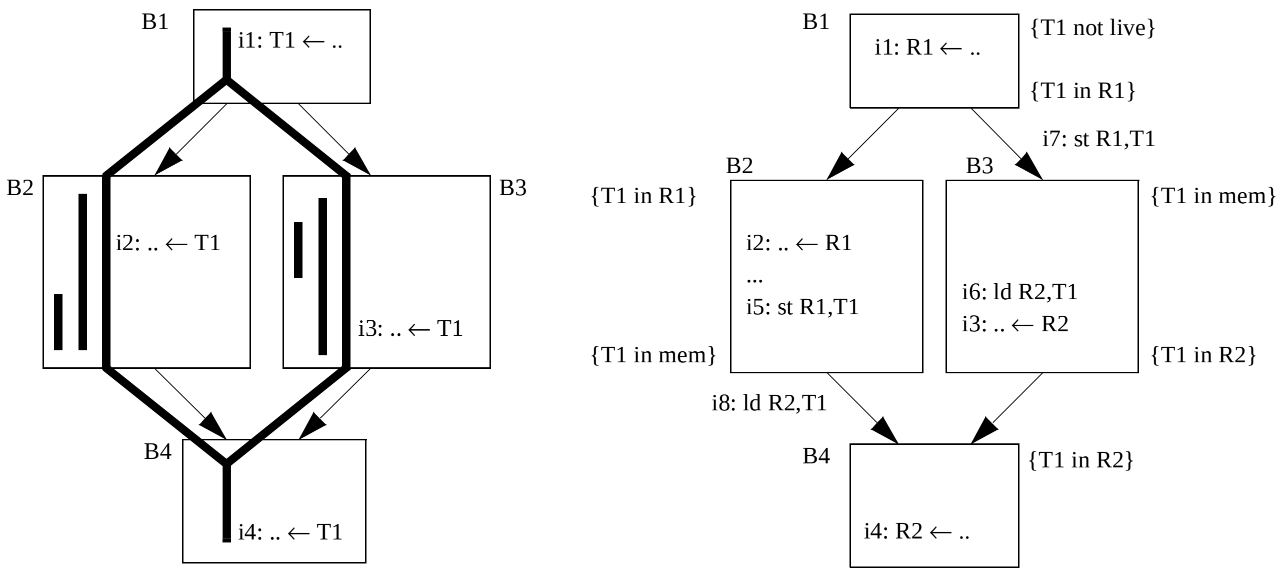
Early on in the design of our binpacking register allocator, we noticed that it is possible to allocate registers to temporaries and rewrite temporary references all in a single linear pass over the program text. When we encounter a temporary *t* for the first time, we interrupt the rewriting process 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. 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 support lifetime splits due to our emphasis on a single allocate/rewrite pass, our allocator supports this optimistic approach naturally.

If the next reference to a spilled temporary *u* is a write, our allocator performs a similar optimistic decision. 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*, and 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. Similar to 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 so that it does not create a store of a temporary *u* currently residing in a register *r* when evicting *u*, if the value for *u* in *r* matches the value for *u* in memory. To perform this optimization, we maintain information about the consistency of



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

(b) The CFG after allocation. Only instructions associated with T1 *are shown. The linear allocation order is* B1-B2-B3-B4. *The allocation assumptions for* T1 *before resolution are shown as sets at the top and bottom of each block.*

Figure 2. Example of conflict resolution at CFG edges. Assume that none of the temporaries contain lifetime holes and that we have only two registers R1 *and* R2. *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.

the value in *r* with respect to the value in *u*'s memory home. Any spill of *u* to or from memory 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.

2.4 Resolution

As we mentioned earlier, the above approach to register allocation comes with a cost. In giving a temporary a second chance and multiple register locations at different intervals in the temporary's lifetime, we can potentially create conflicts in the allocation assumptions at the 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 phi-nodes by a set of equivalent move operations [12]. Figure 2 gives a simple example of resolution.

The linear processing of the CFG can also lead to unnecessary spill loads. Continuing with the example in Figure 2, 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 rewrite in **i3**. This is because the linear ordering assumes that the last action in block **B2** for **T1** left **T1** in memory. 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 advan-

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.

tage 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 2) 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 stack 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 may then be eliminated by subsequent copy propagation and dead-code elimination passes.

Though we do not perform any dataflow analyses during register allocation to minimize the generation or improve the placement of spill code, we do perform, during the resolution phase of our allocator, one dataflow analysis for correctness. If we decided not to insert a store instruction when evicting a temporary (see Section 2.3), we used the fact 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 was 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 linear scan, 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 2.3, a write to *r* clears A_t, and a spill of *t* sets 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 linear scan, 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 we used A_t to inhibit the 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 $p \rightarrow s$ 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.

2.5 Move optimizations

Modern computing systems 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 a graph-coloring register allocator, this is equivalent to adding an interference edge to the caller-saved registers.

In our algorithm, we represent the constraints on register usage by considering the intervals in which a register is free for use as its lifetime holes. A temporary can now fit inside a register's lifetime hole or another temporary's lifetime hole. In order to overcome the problem described above, we allow in our algorithm for a temporary to be assigned to a register with a lifetime hole that is not large enough to contain the entire lifetime. The algorithm heuristically searches for the largest of these insufficiently-large holes. 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 convention, we could insert a spill store, reloading its value the next time we need it through our second-chance mechanism. But it might be true at this point that some other register r_s now contains 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 therefore insert the move now only if we can find an empty register r_s and if evicting *t* from r_t would result in a spill store. We refer to this mechanism as *early second chance*.

Although a move instruction can be 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 peephole pass. The check works as follows: once we have assigned 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.

We have implemented only a limited version of the move elimination optimization. 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 them in the code, they can noticeably degrade the performance of call-intensive programs. Our current implementation performs the move optimization only when the source of a move is already in a register. It would be straightforward to extend our implementation to attempt move optimization after allocation of a general move source.

2.6 Complexity analysis

The conflict resolution step of our algorithm, which we feel is essential for maximizing the quality of the output code, does not have a linear time bound. Its worst-case complexity is dominated by that of the dataflow calculation described above. However, this dataflow analysis can be replaced so that our allocator runs in linear time.

The first two phases of the algorithm, computation of lifetimes and holes, then allocation and rewriting, are manifestly linear.[2] 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 above 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. The common experience with the standard method, however, is that it terminates in two or three iterations at most, which brings its time 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.

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.

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 [14] of the Stanford SUIF compiler system [16]. 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 the central register assignment algorithms. In both passes, for example, we use shared libraries to construct CFGs and perform liveness and loop-depth analysis, with the results attached 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. In each case, register allocation is preceded by dead code elimination and followed by a peephole optimization pass that removes moves that can safely collapse into the preceding or succeeding instruction.

The coloring method used is an implementation of that described by George and Appel [7]. 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. 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 [7] with two exceptions:

- We use a lower-triangular bit matrix, rather than a hash table, to record the adjacency relation of the interference graph.

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.

Benchmark	Instruction counts			Run time (sec)		
	Second-chance binpacking	Graph coloring	Ratio (binpack/GC)	Second-chance binpacking	Graph coloring	Ratio (binpack/GC)
alvinn	5859032035	5850062031	1.002	20.56	20.67	0.995
doduc	1610607538	1565260889	1.029	7.36	7.23	1.018
eqntott	2782873030	2777476231	1.002	6.92	6.90	1.003
espresso	1510435454	1390526882	1.086	3.54	3.34	1.060
fpppp	6775315066	6262634084	1.082	25.79	24.73	1.043
li	9878244999	9694580392	1.019	23.91	24.76	0.966
tomcatv	6531688057	6531662363	1.000	14.29	14.36	0.995
compress	94956007702	91999060755	1.032	281.30	275.79	1.020
m88ksim	1112471957	1101374080	1.010	2.97	2.90	1.024
sort	1030126044	989670114	1.041	4.35	4.02	1.082
wc	1046734	1046722	1.000	0.92	0.91	1.011

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

- 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, and 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. Table 1 presents run-time results both in terms of instruction counts and actual run times. For each metric, we also 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. The target machine for these experiments is a Digital Alpha running Digital UNIX 4.0. Most benchmarks are from the SPEC92 suite, except for *compress* and *m88ksim* (SPEC95) and *sort* and *wc* (UNIX utilities). The instruction count results were obtained using the HALT tool within Machine SUIF to instrument each benchmark after code generation. The run-time results were obtained with the UNIX time command on a lightly-loaded Alpha. Each time is the best of five consecutive runs.

Benchmark	Second-chance binpacking	Graph coloring
alvinn	0%	0%
doduc	0.460%	0.492%
eqntott	0.001%	0.000%
espresso	0.783%	0.148%
fpppp	18.561%	13.397%
li	0%	0%
tomcatv	0%	0%
compress	0%	0%
m88ksim	0.030%	0.045%
sort	1.339%	0.905%
wc	0%	0%

Table 2: 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%".

Overall, our approach produces executables that are of a quality near to those produced by coloring. To help explain the variation in the instruction count results, Table 2 presents a statistic indicating what percentage of the total dynamic instruction count was due to spill code inserted by the register allocator. This counts load, store, and move instructions inserted for allocation candidates only. Five of our benchmarks (*alvinn*, *li*, *tomcatv*, *compress*, and *wc*) had no spill code under either approach. For these applications, the difference in the dynamic instruction counts in Table 1 is entirely due to the lack of move coalescing in our algorithm. We expect that we could remove much of this difference by following register allocation by copy propagation and dead-code elimination optimizations.

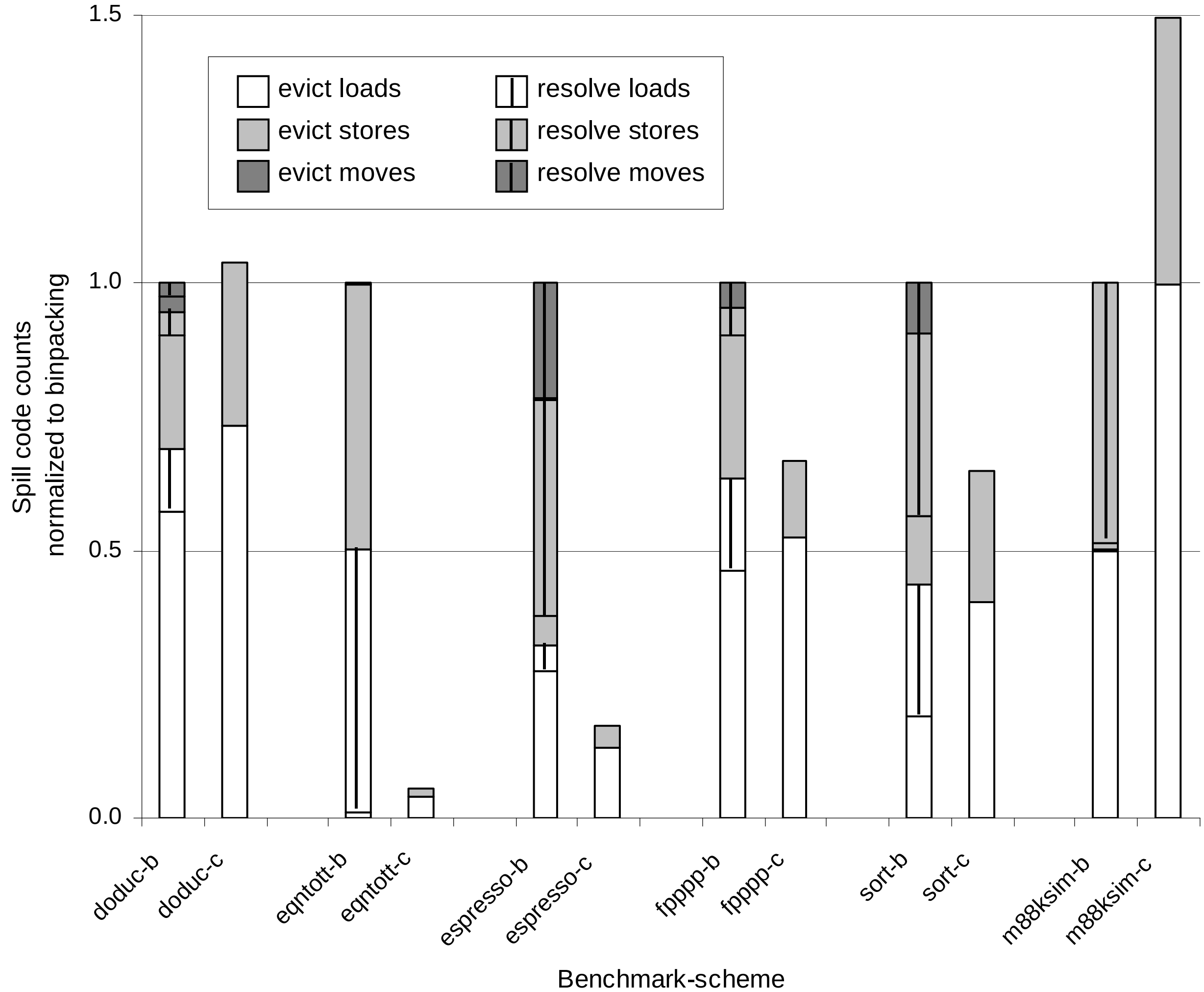


Figure 3. A categorization of the spill code inserted by each allocator. Results for our binpacking approach are labelled with a "-b" while those for coloring are labelled with "-c". For each benchmark, we normalize the counts to the total spill code inserted with binpacking. We have separated the "eviction" spill code inserted during our linear scan and the coloring algorithm's spill phase from the "resolve" spill code inserted during our resolution phase.

For the applications with spill code, Figure 3 presents a detailed look at the composition of the spill code produced both by second-chance binpacking and by graph coloring. In both *doduc* and *m88ksim*, binpacking produces less spill code than coloring. The majority of the difference is due to the insertion of extra spill loads during coloring. Our binpacking produces more spill code than coloring for *eqntott*, *espresso*, *fpppp*, and *sort*. 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 (1445466 vs. 1046734 dynamic instructions) when allocated using two-pass binpacking than it did when allocated with 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

Module (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

Table 3: 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.

live across the call), thus evicting 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 (2783984589 vs. 2782873030 dynamic instructions). The *eqntott* benchmark spends a vast 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 3 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, such as 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 3. 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.

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

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 [13]. 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 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 speed.

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] was the starting point for our work on linear-scan allocation. Binpacking evolved from work done in the production quality compiler-compiler project at CMU [11,17]. 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 [9].

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 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* [8]. 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.

In his recent book, Bob Morgan presents a hybrid approach to register allocation [12]. 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.

5 Conclusions

Linear-scan methods of register allocation are fast and effective. They can enable the interprocedural optimization of large programs, and they 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, and it pays 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. We believe there remain ways of tuning the second-chance binpacking algorithm so that the run-time performance of generated code more uniformly matches that of a coloring allocator.

6 Acknowledgments

We are grateful to Steve Hobbs, Bob Morgan, and August Reinig of Digital Equipment Corporation for their helpful discussions on the use of binpacking in the GEM compiler. We would also like to thank Max Poletto of MIT for his explanation of the use of linear scan in dynamic code generation.

This research was funded in part by a NSF Young Investigator award (grant no. CCR-9457779), a DARPA grant no. NDA904-97-C-0225, and research gifts from AMD, Digital Equipment, HP, Intel, and Microsoft.

7 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, pp. 47–57, 1981.

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

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

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

[8] 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*, pp. 176–191, October 1992.

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

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

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

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

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

[14] 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.

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

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

[17] 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, 1975.