Linear Scan Register Allocation

MASSIMILIANO POLETTO
Laboratory for Computer Science, MIT
and
VIVEK SARKAR
IBM Thomas J. Watson Research Center

We describe a new algorithm for fast global register allocation called *linear scan*. This algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables' live ranges. The linear scan algorithm is considerably faster than algorithms based on graph coloring, is simple to implement, and results in code that is almost as efficient as that obtained using more complex and time-consuming register allocators based on graph coloring. The algorithm is of interest in applications where compile time is a concern, such as dynamic compilation systems, "just-in-time" compilers, and interactive development environments.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; code generation; optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Code optimization, compilers, register allocation

1. INTRODUCTION

Register allocation is an important optimization affecting the performance of compiled code. For example, good register allocation can improve the performance of several SPEC benchmarks by an order of magnitude relative to when they are compiled with poor or no register allocation. Unfortunately, most aggressive global register allocation algorithms are computationally expensive due to their use of the graph coloring framework [Chaitin et al. 1981], in which the interference graph can have a worst-case size that is quadratic in the number of live ranges.

We describe a global register allocation algorithm, called *linear scan*, that is not based on graph coloring. Rather, given the live ranges of variables in a function, the algorithm scans all the live ranges in a single pass, allocating registers to variables in a greedy fashion. The algorithm is simple, efficient, and produces relatively good code. It is useful in situations where both compile time and code quality

This research was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and N66001-96-C-8522. Max Poletto was also supported by an NSF National Young Investigator Award awarded to Frans Kaashoek.
A synopsis of this algorithm first appeared in [Poletto et al. 1997].
Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1999 ACM XXXX-YYYY/ZZ/AAAA-BBBB $CC.DD

are important, such as dynamic compilation systems, just-in-time compilers, and interactive development environments.

We evaluate both the compile-time performance of the linear scan algorithm and the run-time performance of its resulting code. To evaluate the compile-time speed of the algorithm, we compare it to a fast graph coloring allocator used in the tcc dynamic compiler [Poletto et al. 1997]. To further evaluate the quality of the generated code, we implemented the algorithm in the Machine SUIF compiler back-end [Smith 1996; Amarasinghe et al. 1993], and compare the resulting code with the code obtained from an aggressive graph coloring algorithm that performs iterated register coalescing [George and Appel 1996]. In addition, we compare linear scan to second-chance binpacking [Traub et al. 1998], a type of linear scan algorithm that invests more work at compile time in order to produce better code.

The linear scan algorithm is up to several times faster than even a fast graph coloring register allocator that performs no coalescing. Nonetheless, the resulting code is quite efficient: on the benchmarks we studied, it is within 12% as fast as code generated by an aggressive graph coloring algorithm for all but two benchmarks. By comparison, other simple and comparably fast register allocation schemes, such as allocating the k available registers to the k most frequently used variables, result in code that is several times slower.

The rest of the article is organized as follows. Section 2 summarizes related work on global register allocation, while Section 3 outlines the program model and representation assumed in this work. The details of the register allocation algorithm appear in Section 4. Section 5 presents measurements of the algorithm's performance. Finally, Section 6 discusses some extensions to the algorithm and directions for future work, and Section 7 summarizes the main results of this work.

2. RELATED WORK

Global register allocation has been studied extensively in the literature. The predominant approach, first proposed by Chaitin et al. [1981], is to abstract the register allocation problem as a graph coloring problem. Nodes in the graph represent *live ranges* (variables, temporaries, virtual/symbolic registers) that are candidates for register allocation. Edges connect live ranges that *interfere*, i.e., live ranges that are simultaneously live at at least one program point. Register allocation then reduces to the graph coloring problem in which colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color. If the graph is not colorable, some nodes are deleted from the graph until the reduced graph becomes colorable. The deleted nodes are said to be *spilled* because they are not assigned to registers. The basic goal of register allocation by graph coloring is to find a legal coloring after deleting the minimum number of nodes (or more precisely, after deleting a set of nodes with minimum total spill cost).

Chaitin's algorithm also features *coalescing*, a technique that can be used to eliminate redundant moves. When the source and destination of a move instruction do not share an edge in the interference graph, the corresponding nodes can be coalesced into one, and the move eliminated. Unfortunately, aggressive coalescing can lead to uncolorable graphs, in which additional live ranges need to be spilled to memory. More recent work on graph coloring [Briggs et al. 1994; George and Appel 1996] has focused on removing unnecessary moves in a conservative manner so as to avoid introducing spills.

Some simpler heuristic solutions also exist for the global register allocation problem. For example, lcc [Fraser and Hanson 1995] allocates registers to the variables with the highest estimated usage counts, places all others on the stack, and allocates temporary registers within an expression by doing a tree walk.

Linear scan can be viewed as a global extension of a special class of local register allocation algorithms that have been considered in the literature [Freiburghouse 1974; Hsu et al. 1989; Fraser and Hanson 1995; Motwani et al. 1995], which in turn take their inspiration from an optimal off-line replacement algorithm that was studied for virtual memory [Belady 1966].

Since our original description of the linear scan algorithm in Poletto et al. [1997], Traub et al. have proposed a more complex linear scan algorithm, which they call *second-chance binpacking* [Traub et al. 1998]. This algorithm is an evolution and refinement of *binpacking*, a technique used for several years in the DEC GEM optimizing compiler [Blickstein et al. 1992]. At a high level, the binpacking schemes are similar to linear scan, but they invest more time in compilation in an attempt to generate better code. The second-chance binpacking algorithm both makes allocation decisions and rewrites code in one pass. The algorithm allows a variable's lifetime to be split multiple times, so that the variable resides in a register in some parts of the program and in memory in other parts. It takes a lazy approach to spilling, and never emits a store if a variable is not live at that particular point or if the register and memory values of the variable are consistent. At every program point, if a register must be used to hold the value of a variable v_1, but v_1 is not currently in a register and all registers have been allocated to other variables, the algorithm evicts a variable v_2 that is allocated to a register. It tries to find a v_2 that is not currently live (to avoid a store of v_2), and that will not be live before the end of v_1's live range (to avoid evicting another variable when both v_1 and v_2 become live).

Binpacking can emit better code than linear scan, but it does more work at compile time. Unlike linear scan, binpacking keeps track of the "lifetime holes" of variables and registers (intervals when a variable maintains no useful value, or when a register can be used to store a value), and maintains information about the consistency of the memory and register values of a reloaded variable. The algorithm analyzes all this information whenever it makes allocation or spilling decisions. Furthermore, unlike linear scan, it must perform an additional "resolution" pass to resolve any conflicts between the nonlinear structure of the control flow graph and the assumptions made during the linear register allocation pass. Section 5 compares the performance of the two algorithms and of the code that they generate.

3. PROGRAM MODEL

Throughout the article, we assume a program intermediate representation that consists of RTL-like quads or pseudo-instructions. Register candidates (live ranges) are represented by an unbounded set of variable names or "virtual registers." Arithmetic operations are performed directly on these virtual registers; no load/store instructions are necessary for accessing virtual registers. By convention, variables are not live on entry to the start node in the flow graph; the initialization of procedure parameters is captured by explicit assignments within the start node.

No variable renaming or live range splitting is performed by our linear scan algorithm. It may be beneficial to perform a renaming phase (such as renaming into "webs" [Muchnick 1997] or computing the "right number of names" [Auslander and Hopkins 1982]) as an optional prepass to our linear scan algorithm. Further study is required to determine the extent to which the compile-time overhead of an extra renaming phase is justified by its accompanying run-time improvement.

The linear scan algorithm assumes that the intermediate representation pseudo-instructions are numbered according to some order. One possible ordering is that in which the pseudo-instructions appear in the intermediate representation. Another is depth-first ordering, the reverse of the order in which nodes are last visited in a preorder traversal of the flow graph [Aho et al. 1986]. Throughout the rest of this article we use depth-first order. The choice of instruction ordering does not affect the correctness of the algorithm, but it may affect the quality of allocation. We discuss alternative orderings in Section 6.

Central to the linear scan algorithm is the notion of a *live interval*. Given some numbering of the intermediate representation, $[i, j]$ is said to be a live interval for variable v if there is no instruction with number $j' > j$ such that v is live at j', and there is no instruction with number $i' < i$ such that v is live at i'. This information is a conservative approximation of live ranges: there may be subranges of $[i, j]$ in which v is not live, but they are ignored. The "trivial" live interval for any variable v is $[1, N]$, where N is the number of pseudo-instructions in the intermediate representation: this live interval is correct and takes no time to compute, but it also yields no information. All other live intervals lie on the spectrum between the trivial live interval and accurate live interval information. The order chosen for numbering pseudo-instructions influences the extent and accuracy of live intervals, and hence the quality of register allocation, but the definition of live intervals does not rely on or make assumptions about a particular numbering.

4. THE LINEAR SCAN ALGORITHM

Given live variable information (obtained, for example, via data-flow analysis [Aho et al. 1986]), live intervals can be computed easily with one pass through the intermediate representation. Interference among live intervals is captured by whether or not they overlap. Given R available registers and a list of live intervals, the linear scan algorithm must allocate registers to as many intervals as possible, but such that no two overlapping live intervals are allocated to the same register. If $n > R$ live intervals overlap at any point, then at least $n - R$ of them must reside in memory.

4.1 Details

The number of overlapping intervals changes only at the start and end points of an interval. Live intervals are stored in a list that is sorted in order of increasing start point. Hence, the algorithm can quickly scan forward through the live intervals by skipping from one start point to the next.

At each step, the algorithm maintains a list, *active*, of live intervals that overlap the current point and have been placed in registers. The *active* list is kept sorted in order of increasing end point. For each new interval, the algorithm scans *active* from beginning to end. It removes any "expired" intervals—those intervals that no

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Fig. 1. Linear scan register allocation. Indentation denotes nesting level. We assume that live intervals (including *startpoint* and *endpoint* information) have been computed by a prior liveness analysis phase.

longer overlap the new interval because their end point precedes the new interval's start point—and makes the corresponding register available for allocation. Since *active* is sorted by increasing end point, the scan needs to touch exactly those elements that need to be removed, plus at most one: it can halt as soon as it reaches the end of *active* (in which case *active* remains empty) or encounters an interval whose end point follows the new interval's start point.

The length of the *active* list is at most R. The worst case scenario is that *active* has length R at the start of a new interval and no intervals from *active* are expired. In this situation, one of the current live intervals (from *active* or the new interval) must be spilled. There are several possible heuristics for selecting a live interval to spill. The heuristic described in this paper is based on the remaining length of live intervals. Our algorithm spills the interval that ends last, furthest away from the current point. We can find this interval quickly because *active* is sorted by increasing end point: the interval to be spilled is either the new interval or the last interval in *active*, whichever ends later. In straight-line code, and when each live interval consists of exactly one definition followed by one use, this heuristic produces code with the minimal possible number of spilled live ranges [Belady 1966; Motwani et al. 1995]. Although in our case a live interval may cover arbitrarily many definitions and uses spread over different basic blocks, the heuristic still appears to

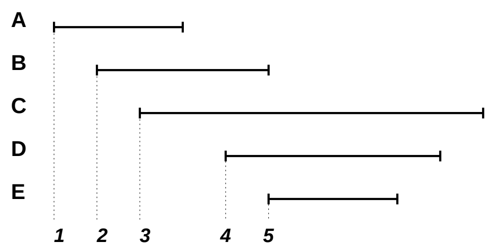

Fig. 2. An example set of live intervals. Letters on the left are variable names; the corresponding live intervals appear to the right. Numbers in italics refer to steps in the linear scan algorithm described in the text.

work well. Figure 1 contains the pseudocode for the linear scan algorithm with this heuristic. All results in Section 5 are also based on this heuristic.

4.2 An Example

Consider, for example, the live intervals in Figure 2 for the case when the number of available registers is $R = 2$. The algorithm performs allocation decisions 5 times, once per live interval, as denoted by the italicized numbers at the bottom of the figure. By the end of step *2*, $active = \langle A, B \rangle$ and both A and B are therefore in registers. At step *3*, three live intervals overlap, so one variable must be spilled. The algorithm therefore spills C, the one whose interval ends furthest away from the current point, and does not change *active*. As a result, at step *4*, A is expired from *active*, making a register available for D, and at step *5*, B is expired, making a register available for E. Thus, in the end, C is the only variable not allocated to a register. Had the algorithm not spilled the longest interval, C, at step *3*, both one of A and B and one of D and E would have been spilled to memory.

4.3 Complexity

Let V be the number of variables (live intervals) that are candidates for register allocation, and R be the number of registers available for allocation. As can be seen from the pseudocode in Figure 1, the length of *active* is bounded by R, so the linear scan algorithm takes $O(V)$ time if R is assumed to be a constant.

Since R can be large in some current or future processors, it is worthwhile understanding how the complexity depends on R. Recall that the live intervals in *active* are sorted in order of increasing endpoint. The worst-case execution time complexity of the linear scan algorithm is dictated by the time taken to insert a new interval into *active*. If a balanced binary tree is used to search for the insertion point, then the insertion takes $O(\log R)$ time and the entire algorithm takes $O(V \times \log R)$ time. An alternative is to do a linear search for the insertion point, which takes $O(R)$ time, thus leading to a worst case complexity of $O(V \times R)$ time. This is asymptotically slower than the previous result, but may be faster for moderate values of R because the data structures involved are much simpler. The implementations evaluated in Section 5 use a linear search.

5. EVALUATION

This section evaluates linear scan register allocation in terms of both compile-time performance and the quality of the resulting code.

5.1 Methodology

We use two different infrastructures, one primarily to measure compile-time performance, and one primarily to measure the run-time performance of the generated code.

5.1.1 *The* ICODE *Infrastructure.* A convincing benchmark of compile-time performance requires that the algorithm be implemented as part of a compiler that is already well-tuned for efficient compile times. As a result, we implemented the algorithm in ICODE, a runtime system of the `tcc` dynamic compiler [Poletto et al. 1999]. `tcc` is an implementation of 'C, an extension to ANSI C that enables dynamic code generation. ICODE is an optimizing dynamic code generation system that produces good quality code with low compile-time overhead (approximately 600 cycles per generated instruction).

We use two sets of benchmarks to evaluate our ICODE implementation. The first is the same as that used in previous experimental studies with ICODE: it consists of several dynamic code kernels, such as numerical methods, matrix multiplication, sorting, etc. For each of these benchmarks, we compare linear scan register allocation against (1) a well-tuned graph coloring algorithm and (2) a simple "usage count" register allocation scheme. The graph coloring algorithm tries to be fast without overly penalizing code quality: it does not do coalescing, but takes reference counts into consideration when removing nodes from the interference graph. The "usage count" algorithm allocates the k available registers to the k variables and compiler-generated temporaries with the highest estimated usage counts, and places all others on the stack.

The second set of benchmarks consists of pathological programs that perform no useful computation but have huge numbers of simultaneously live variables that make register allocation difficult. We use these benchmarks to compare the performance of graph coloring and linear scan as the size of the allocation problem increases.

All experiments were made on an unloaded Sun Ultra 2 Model 2170 workstation with 384MB of main memory and a 168MHz UltraSPARC-I CPU. Times were the sum of system and user times reported by the UNIX `getrusage` system call. Values for each benchmark were obtained by taking the mean of ten trials. The standard deviation for each set of trials was negligible. The value for each trial was computed by timing a large number of runs (so as to provide several seconds of granularity), and dividing the result by the number of runs.

5.1.2 *The SUIF Infrastructure.* Since the 'C benchmarks discussed above are all relatively small, their run-time performance is similar for all the register allocation algorithms. In order to measure the effect of linear scan on the performance of larger programs, we implemented it in Machine SUIF [Smith 1996], an optimizing scalar back end infrastructure for SUIF [Amarasinghe et al. 1993]. We used this implementation to compile various SPEC benchmarks (from both the SPEC92 and

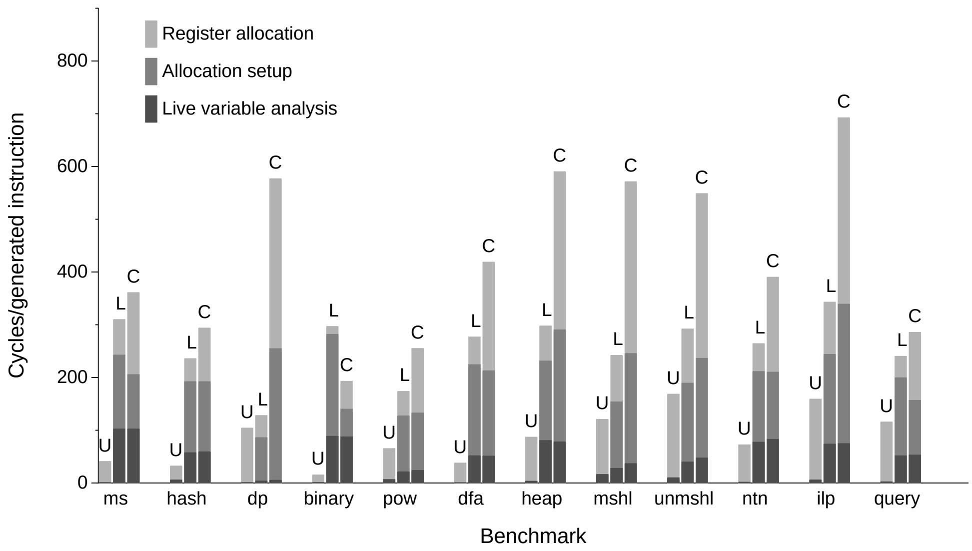

Fig. 3. Register allocation overhead for dynamic code (‘C) kernels. U denotes a simple algorithm based on usage counts. L denotes linear scan. C denotes graph coloring.

SPEC95 suites) and two UNIX utilities. As before, we compare linear scan register allocation against a graph coloring algorithm and the simple algorithm based on usage counts. We also compare it against second-chance binpacking [Traub et al. 1998]. The graph coloring allocator is an implementation of iterated register coalescing [George and Appel 1996] developed at Harvard. For completeness, we also report the compile-time performance of the SUIF implementation of binpacking and linear scan in Section 5.2.2, even though the underlying SUIF infrastructure has not been designed for efficient compile times.

All benchmarks were compiled with SUIF and Machine SUIF. Measurements are the user time from the best of ten runs on an unloaded DEC Alpha workstation with a 500MHz Alpha 21164 processor and 128MB of RAM.

5.2 Compile-Time Performance

5.2.1 ICODE *Implementation.* Figure 3 illustrates the overhead of register allocation for the dynamic code kernels described in Section 5.1.1. The vertical axis measures compilation overhead, in cycles per generated instruction. Larger values indicate larger overhead. The horizontal axis of the figure denotes different benchmarks written in ‘C. For each benchmark, there are three bars: U refers to the usage count algorithm; L refers to linear scan register allocation; C refers to graph coloring.

Each bar contains up to three different regions:

(1) **Live variable analysis:** refers to traditional iterative live variable analysis, and hence does not appear in the U column.
(2) **Allocation setup:** refers to work necessary prior to register allocation. It does not apply to U. In the case of L, it refers to the construction of live intervals

Table I. Allocation Times for Linear Scan and Binpacking

File (Benchmark)	Time in seconds		Ratio
	Linear scan	Binpacking	(Binpacking / linear scan)
swim.f (swim)	0.42	1.07	2.55
xllist.c (li)	0.31	0.60	1.94
xleval.c (li)	0.14	0.29	2.07
tomcatv.f (tomcatv)	0.19	0.48	2.53
compress.c (compress)	0.14	0.32	2.29
cvrin.c (espresso)	0.61	1.14	1.87
backprop.c (alvinn)	0.07	0.19	2.71
fpppp.f (fpppp)	3.35	4.26	1.27
twldrv.f (fpppp)	1.70	3.49	2.05

by coarsening live variable information obtained through live variable analysis. In the case of C, it refers to construction of the interference graph.

(3) **Register allocation:** in the case of U, it involves sorting variables by usage count, and allocating registers to the most used ones until none are left. In the case of L, it refers to linear scan of the live intervals. In the case of C, it refers to coloring the interference graph.

Liveness analysis and allocation setup for the U case are essentially null function calls. Small positive values for these two phases, as well as small differences in the live variable analysis overheads in the L and C cases, are due to slight variability in the `getrusage` measurements. Times for individual compilation phases were obtained by repeatedly interrupting compilation after the phase of interest, subtracting the time required up to the previous phase, and dividing by the number of (interrupted) compiles.

The figure indicates that linear scan allocation (L) can be considerably faster than even a simple and fast graph coloring algorithm (C). In particular, although creating live intervals from live variable information is roughly similar to building an interference graph from live variable information, linear scan of live intervals is always much faster than coloring the interference graph. The one benchmark in which graph coloring is faster than linear scan is binary. In this case, the code uses very few variables but consists of many basic blocks, so it is faster to build the small interference graph than to extract live intervals from liveness information at each basic block. However, note that even for binary the actual time spent on register allocation is smaller for linear scan (L) than for graph coloring (C).

5.2.2 *SUIF Implementation.* Table I compares the compile-time performance of the SUIF implementation of binpacking and linear scan on representative files from the benchmark set. We do not present data for graph coloring: [Traub et al. 1998] and Section 5.2.3 provide convincing evidence that both binpacking and linear scan are much faster than graph coloring, especially as the number of register candidates grows.

The times in Table I refer to only the core allocation routines: they include neither setup activities such as CFG construction and liveness analysis, nor any compilation phase after allocation. In most cases, linear scan is roughly two to three times faster than binpacking.

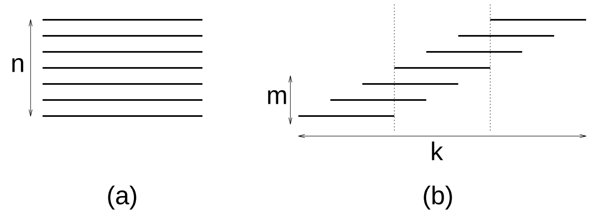

Fig. 4. Two types of pathological programs.

These results, however, underrepresent the difference between the two algorithms. For simplicity, the linear scan implementation uses the binpacking routine for computing "lifetime holes" [Traub et al. 1998]. However, linear scan does not need or use full information on lifetime holes—it just considers the start and end of each variable's live interval. As a result, an aggressive linear scan implementation could be considerably faster. For example, if one does not count lifetime hole computation, the compilation overhead for fpppp.f is $2.79s$ with binpacking and $1.88s$ with linear scan, and that of twldrv.f is $2.28s$ with binpacking and $0.49s$ with linear scan.

5.2.3 *Pathological Cases.* We also employed the ICODE framework used in Section 5.2.1 to compile pathological programs intended to stress register allocators. We compiled programs with two different kinds of structure, as illustrated in Figure 4. One kind, labeled (a) in the figure, contains some number, n, of overlapping live intervals (simultaneously live variables). The other kind, labeled (b), contains k staggered "sets" of live intervals in which no more than m live intervals overlap.

Figure 5 illustrates the overhead of graph coloring and linear scan as a function of the number n of overlapping live intervals in code of type (a). Both axes are logarithmic. The horizontal axis indicates problem size; the vertical axis indicates time. Although the costs of graph coloring and linear scan are comparable when the number of overlapping live intervals is small, linear scan scales much more gracefully to large problem sizes. With 512 simultaneously live variables, linear scan is over 600 times faster than graph coloring. Unlike linear scan, graph coloring appears to suffer from the $O(n^2)$ time required to build and color the interference graph. Importantly, the reported overhead is for the entire code generation process—not just allocating registers, but also setting up the intermediate representation, computing live variables, and generating code, so both algorithms share a common fixed cost that reduces the relative performance gap between them. Furthermore, the code generated by both allocators for this pathological case contains the same number of spills.

Figure 6 compares the overhead of graph coloring and linear scan for programs with live interval patterns of type (b). As in the previous experiment, linear scan in this case generates the same number of spills as graph coloring. Again, the axes are logarithmic and the vertical axis indicates time. The horizontal axis denotes the number of successive staggered sets of live intervals, k in Figure 4(b). Different curves denotes different numbers of simultaneously live variables (m in Figure 4(b): for example, "Linear Scan (m=24)" refers to linear scan allocation with m = 24. With increasing k, the overhead of graph coloring grows more quickly than that

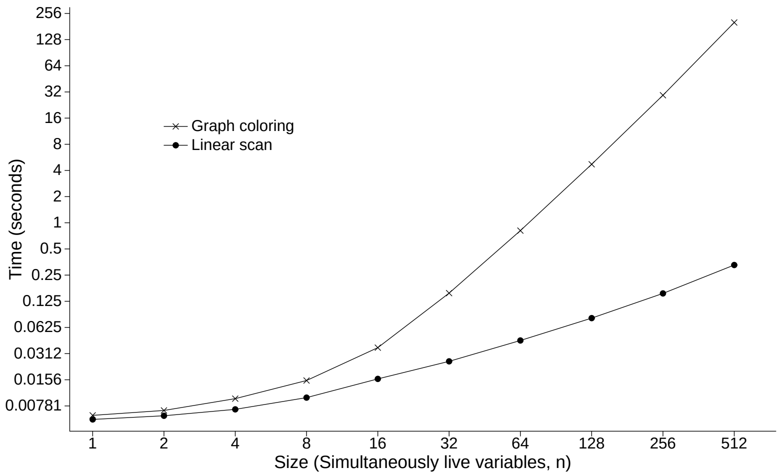

Fig. 5. Overhead of graph coloring and linear scan as a function of the number of simultaneously live variables for programs of type (a).

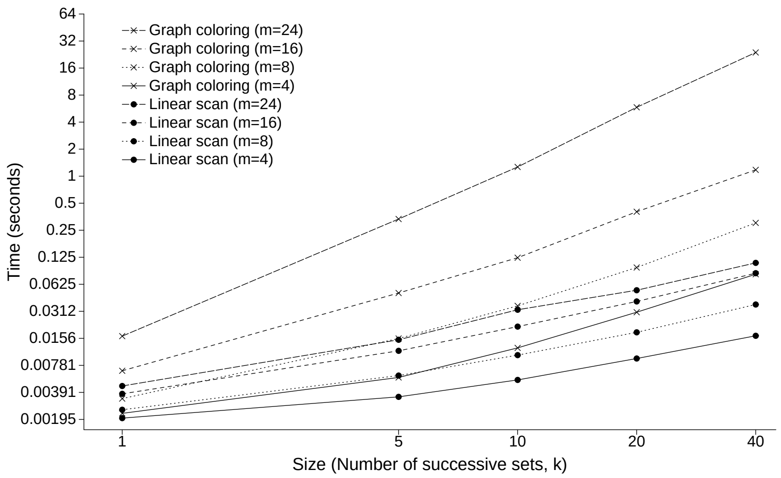

Fig. 6. Overhead of graph coloring and linear scan as a function of program size for programs of type (b). The horizontal axis denotes the number of staggered sets of intervals (k in Figure 4(b)). Different curves denote values for different numbers of simultaneously live variables (m in Figure 4(b)).

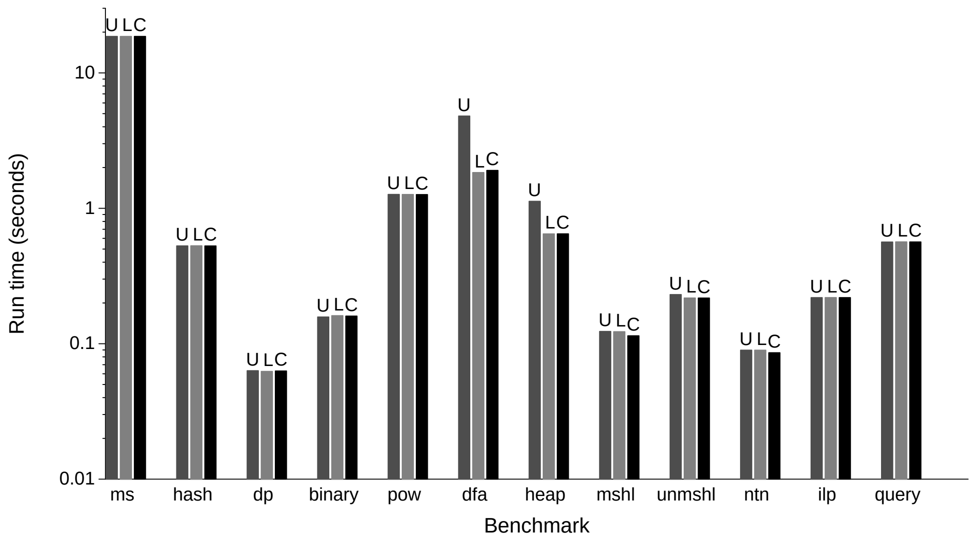

Fig. 7. Run time of 'C benchmarks compiled with different register allocation algorithms. U denotes the simple scheme based on usage counts. L denotes linear scan. C denotes graph coloring.

of linear scan. Moreover, the vertical space between graph coloring curves for increasing m grows more quickly than for the corresponding linear scan curves. This data is consistent with the results in Figure 5: the performance of graph coloring degrades as the number of simultaneously live variables increases.

5.3 Run-Time Performance

5.3.1 ICODE *Implementation.* Figure 7 shows the run-time performance of code compiled with the ICODE implementation (described in Section 5.1.1) of the algorithms. As before, the horizontal axis denotes different benchmarks, and for each benchmark the different bars denote different register allocation algorithms, labeled as in Figure 3. The vertical axis is logarithmic, and indicates run time in seconds. Unfortunately, these dynamic code kernels are small, and do not have enough register pressure to illustrate the differences among the allocation algorithms. The three algorithms generate code of similar quality for all benchmarks other than dfa and heap. In these two cases, the code emitted by the simple allocator based on usage count is considerably slower than that created by graph coloring or linear scan.

5.3.2 *SUIF Implementation.* Figure 8 presents the run time of several large benchmarks compiled with the SUIF implementation of the algorithms. Once again, the horizontal axis denotes different benchmarks, and the logarithmic vertical axis measures run time in seconds. In addition to the three algorithms (U, L, and C) measured so far, we also present data for second-chance binpacking [Traub et al. 1998], which we label B. Table II contains the same data, and also provides the ratio of the run time of each benchmark compiled with each register allocation method relative to the run time of that benchmark compiled with graph coloring.

The measurements in Figure 8 and Table II indicate that linear scan makes a

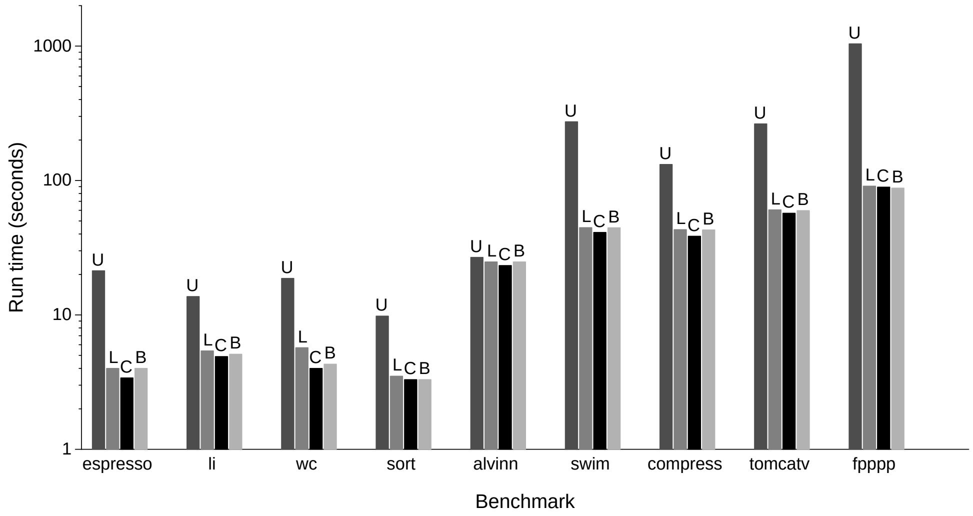

Fig. 8. Run times of static C benchmarks compiled with different register allocation algorithms. U, L, and C are as before. B denotes second-chance binpacking.

Table II. Run Times of Benchmarks, as a Function of the Register Allocation Algorithm Used when Compiling Them

Benchmark	**Time in seconds (ratio to graph coloring)**			
	Usage counts	Linear scan	Graph coloring	Binpacking
espresso	21.3 (6.26)	4.0 (1.18)	3.4 (1.00)	4.0 (1.18)
compress	131.7 (3.42)	43.1 (1.12)	38.5 (1.00)	42.9 (1.11)
li	13.7 (2.80)	5.4 (1.10)	4.9 (1.00)	5.1 (1.04)
alvinn	26.8 (1.15)	24.8 (1.06)	23.3 (1.00)	24.8 (1.06)
tomcatv	263.9 (4.62)	60.5 (1.06)	57.1 (1.00)	59.7 (1.05)
swim	273.6 (6.66)	44.6 (1.09)	41.1 (1.00)	44.5 (1.08)
fpppp	1039.7 (11.64)	90.8 (1.02)	89.3 (1.00)	87.8 (0.98)
wc	18.7 (4.67)	5.7 (1.43)	4.0 (1.00)	4.3 (1.07)
sort	9.8 (2.97)	3.5 (1.06)	3.3 (1.00)	3.3 (1.00)

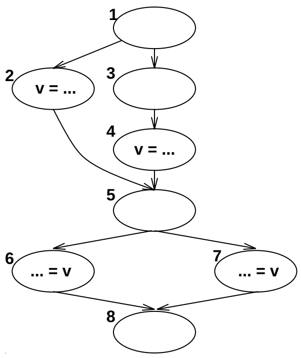

Fig. 9. An acyclic flow graph. Nodes are labeled with their depth-first numbers.

fair performance tradeoff. It is considerably simpler and faster than graph coloring and binpacking, yet it usually generates code that runs within 10% of the speed of that generated by the two more complicated algorithms, and several times faster than that generated by the simple usage count allocator.

6. DISCUSSION

This section addresses various extensions and issues related to linear scan allocation. In particular, we describe a fast algorithm for conservative (approximate) live interval analysis, discuss the effect of different flow graph numberings and spilling heuristics, mention some architectural considerations, and outline possible future refinements to linear scan allocation.

6.1 Fast Live Interval Analysis

Figure 3 shows that most of the overhead of linear scan register allocation is due to live variable analysis and "allocation setup," the coarsening of live variable information into live intervals. As a result, we have experimented with an alternative algorithm that trades accuracy for speed, and quickly builds a conservative approximation of live intervals without requiring full iterative live variable analysis.

We call this algorithm "SCC-based liveness analysis" because it is based on the decomposition of the flow graph into strongly connected components. It relies on two simple observations, which we present here without proof. First, consider an *acyclic* flow graph in which nodes are numbered in depth-first order (also known as "reverse postorder" [Aho et al. 1986]), as shown in the example in Figure 9. Recall that this order is the reverse of the order in which nodes are last visited, or "finished" [Cormen et al. 1990], in a preorder traversal of the graph. If the assignment to a variable v with the smallest depth-first number (DFN) has DFN i, and the use with the greatest DFN has DFN j, then $[i, j]$ is a live interval of v. For example, in Figure 9, a conservative live interval of v is $[2, 7]$. The second observation pertains to cyclic flow graphs: when all the definitions and uses of a variable v appear within a single strongly connected component, C, of the flow graph, the live interval of v will span at most exactly C.

As a result, we can compute conservative live intervals as follows. (1) Compute

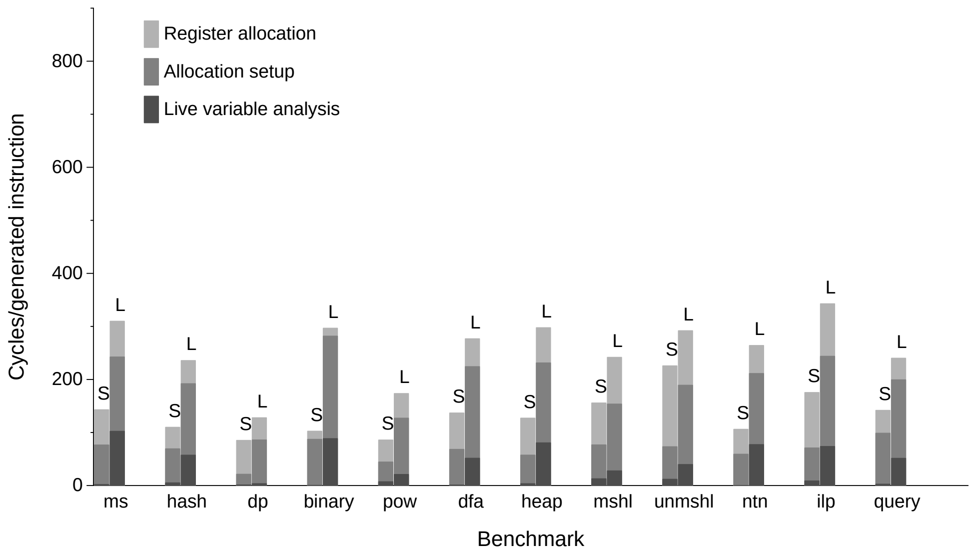

Fig. 10. Comparison of register allocation overhead of linear scan with full live variable analysis (L) and SCC-based liveness analysis (S).

Table III. Run Time of Programs Compiled with Linear Scan Allocation, as a Function of Liveness Analysis Technique

Benchmark	Time in seconds (ratio to graph coloring)	
	SCC-based analysis	Full liveness analysis
espresso	22.7 (6.68)	4.0 (1.18)
compress	134.4 (3.49)	43.1 (1.12)
li	14.2 (2.90)	5.4 (1.10)
alvinn	40.2 (1.73)	24.8 (1.06)
tomcatv	290.8 (5.09)	60.5 (1.06)
swim	303.5 (7.38)	44.6 (1.09)
fpppp	484.7 (5.43)	90.8 (1.02)
wc	23.2 (5.80)	5.7 (1.43)
sort	10.6 (3.21)	3.5 (1.06)

SCCs of the flow graph, and for each SCC, construct the set of variables used or defined in it. Also obtain each SCC's DFN in the (acyclic) SCC graph. (2) Traverse the SCCs once, extending the live interval of each variable v to $[i, j]$, where i and j are, respectively, the smallest and largest DFNs of any SCCs that use or define v.

This algorithm is appealing because it is simple and it minimizes expensive bit-vector operations common in live variable analysis. The improvements in compile-time relative to linear scan are impressive, as illustrated in Figure 10.

Unfortunately, however, the quality of generated code suffers from this approximate analysis. The difference is minimal for the small ‘C benchmarks presented in Section 5.1.1, but becomes prohibitive for large benchmarks. Table III compares the run time of applications compiled with full live variable analysis to that of ap-

Table IV. Run Time of Programs Compiled with Linear Scan Allocation, as a Function of Flow Graph Numbering

Benchmark	**Time in seconds**	
	Depth-first	Linear (layout)
espresso	4.0	4.0
compress	43.3	43.6
li	5.3	5.5
alvinn	24.9	25.0
tomcatv	60.9	60.4
swim	44.8	44.4
fpppp	90.8	91.1
wc	5.7	5.8
sort	3.5	3.6

Table V. Run Time of Programs Compiled with Linear Scan Allocation, as a Function of Spilling Heuristic

Benchmark	**Time in seconds**	
	Interval length	Interval weight
espresso	4.0	4.0
compress	43.1	43.0
li	5.4	5.4
alvinn	24.8	24.8
tomcatv	60.5	60.2
swim	44.6	44.6
fpppp	90.8	198.6
wc	5.7	5.7
sort	3.5	3.5

plications compiled with SCC-based liveness analysis. These results indicate that SCC-based liveness analysis may be of interest for quickly compiling small functions, but that it is not suitable as a replacement for full live variable analysis in large programs.

6.2 Numbering Heuristics

As mentioned in Section 3, the definition of live intervals used in linear scan allocation holds for any numbering of flow graph nodes, not just the depth-first numbering discussed so far.

We have used depth-first order in the paper because it is the most natural, and it supports SCC-based liveness analysis. Another reasonable alternative is linear, or layout, order, i.e. the order in which the pseudo-instructions appear in the intermediate representation. As shown in Table IV, linear and depth-first order produce roughly similar code for our set of benchmarks.

6.3 Spilling Heuristics

The spilling heuristic presented in Section 4 uses interval length. We also considered an alternative spilling heuristic based on interval weight, or estimated usage count. In this case, the algorithm spills the interval with the least estimated usage count among the new interval and the intervals in *active*.

Table V compares the run time of programs compiled using interval length and interval weight spilling heuristics. In general, the results are similar; only in one benchmark, fpppp, does the interval length heuristic significantly outperform interval weight. Of course, the relative performance of the two heuristics depends entirely on the structure of the program being compiled. The interval length heuristic has the additional advantage that it is slightly simpler, since it does not require maintaining usage count information.

6.4 Architectural Considerations

Many machines place restrictions on the use of registers: for instance, only certain registers may be used to pass arguments or return results, or certain operations must target specific registers.

Operations that target specific registers can be handled by pre-allocating the register candidates that are targets of these instructions, and modifying the allocation algorithm to take the pre-allocation into account. In the case of linear scan, if the scan encounters a pre-allocated live interval, it must spill (or assign to a different register) the interval in *active*, if any, that already uses the register of the pre-allocated interval. However, because live intervals are so coarse, it is possible that two intervals that both need to use a particular register overlap: in this case, the best solution is to extend linear scan so that a variable may reside in different locations throughout its lifetime, as in binpacking [Traub et al. 1998].

The issue of when to use caller-saved registers is solved elegantly by the binpacking algorithm, which extends the concept of lifetime holes to physical registers [Traub et al. 1998]. The lifetime hole of a register is any region in which that register is available for allocation; the lifetime holes of caller-saved registers simply do not include function calls. Linear scan does not use lifetime holes. The simplest solution is to use all registers, and insert saves and restores where appropriate around function calls after register allocation. Another solution is to actually introduce the binpacking concept of register lifetime hole, and to allocate a live interval to a register only if it fits entirely within the register's lifetime hole. We use the former solution in the ICODE implementation (Section 5.1.1), and the latter solution in the SUIF implementation (Section 5.1.2).

6.5 Optimizations

Since it is so simple, linear scan allocation lends itself to several refinements and optimizations. The most beneficial in terms of code quality is probably live interval splitting. Splitting does not involve changes to linear scan itself, but only to the definition of live intervals. With splitting, a variable has one live interval for each region of the flow graph throughout which it is uninterruptedly live, rather than one interval for the entire flow graph. This definition takes advantage of holes in variable lifetimes, and is analogous to binpacking without the second-chance technique [Traub et al. 1998]. Past work on renaming scalar variables, including renaming into SSA form [Cytron et al. 1989], can be useful in subsuming much of the splitting that might be useful for register allocation.

Another possible optimization is coalescing of register moves. If the live interval of a variable v_1 ends where the live interval of another variable, v_2, begins, and the program at that point contains a copy $v_2 \leftarrow v_1$, then v_2 can be assigned v_1's register, and if v_2 is not subsequently spilled, the move can be eliminated after register allocation. It is not difficult to extend the routine ExpireOldIntervals in Figure 1 to enable this optimization. However, in order to be effective, move coalescing depends on live interval splitting: without splitting, the opportunities for coalescing are few and can occur only outside of loops.

We have considered these and other optimizations, but have not yet implemented them. The key advantage of the linear scan algorithm is that it is fast and simple, yet can produce relatively good code. Additions to linear scan would make it slower and more complicated, and may not improve the generated code much. If one is willing to sacrifice some compile-time performance to obtain better code, then the second-chance binpacking algorithm might be a good solution.

7. CONCLUSION

The linear scan algorithm is a fast and simple technique for global register allocation. Rather than coloring an interference graph, the algorithm allocates registers by making a single pass over coarse live interval information.

Measurements indicate that the linear scan algorithm is significantly faster than graph coloring algorithms, and that it generally emits code that runs within approximately 10% of the speed of that generated by an aggressive graph coloring algorithm.

Linear scan register allocation is being used as part of the tcc dynamic compilation system. It should also be well suited to other applications where compile time and code quality are important, such as "just-in-time" compilers and interactive development environments.

ACKNOWLEDGMENTS

We thank Omri Traub, Glenn Holloway, and Michael Smith at Harvard University for their willingness to discuss their binpacking algorithm and to share their Machine SUIF register allocation infrastructure. We are grateful to Frans Kaashoek at MIT for his comments and feedback, and for his support of this research. We also thank the anonymous referees for their helpful comments, which improved the evaluation and discussion sections.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

Amarasinghe, S. P., Anderson, J. M., Lam, M. S., and Lim, A. W. 1993. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing.* Portland, OR.

Auslander, M. and Hopkins, M. 1982. An overview of the PL.8 compiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.* 22–31.

Belady, L. A. 1966. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal 5,* 2, 78–101.

Blickstein, D., Craig, P., Davidson, C., Faiman, R., Glossop, K., Grove, R., Hobbs, S., and Noyce, W. 1992. The GEM optimizing compiler system. *Digital Equipment Corporation Technical Journal 4,* 4, 121–135.

Briggs, P., Cooper, K., and Torczon, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16,* 3 (May), 428–455.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Computer Languages 6*, 47–57.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1989. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages.* Austin, TX, 25–35.

FRASER, C. W. AND HANSON, D. R. 1995. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA.

FREIBURGHOUSE, R. A. 1974. Register allocation via usage counts. *Communications of the ACM 17,* 11 (November), 638–642.

GEORGE, L. AND APPEL, A. 1996. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems 18,* 3 (May), 300–324.

HSU, W.-C., FISCHER, C. N., AND GOODMAN, J. R. 1989. On the minimization of loads and stores in local register allocation. *IEEE Transactions on Software Engineering 15,* 10 (October), 1252–1260.

MOTWANI, R., PALEM, K. V., SARKAR, V., AND REYEN, S. 1995. Combining Register Allocation and Instruction Scheduling (Technical Summary). Tech. rep., Courant Institute, New York University. July. TR 698.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann, San Francisco, CA.

POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.* Las Vegas, NV, 109–121.

POLETTO, M., HSIEH, W. C., ENGLER, D. R., AND KAASHOEK, M. F. 1999. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems.* (To appear).

SMITH, M. 1996. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop.* Stanford, CA, 14–25. `http://www.eecs.harvard.edu/machsuif`.

TRAUB, O., HOLLOWAY, G., AND SMITH, M. D. 1998. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.*

Received February 1998; revised July 1998; accepted September 1998